Head First
Android Development



Jonathan Simon

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

Head First Android Development

by Jonathan Simon
Copyright © 2011 Jonathan Simon. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly Media books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Series Creators:	Kathy Sierra, Bert Bates
Editor:	Brian Sawyer
Cover Designers:	Karen Montgomery
Production Editor:	TK
Indexer:	TK
Proofreader:	TK
Page Viewers:	Felisa

Printing History:

October 2011: First Edition.

Felisa



The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. The *Head First* series designations, *Head First Android Development* and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and the authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-1-449-39330-4

[M]

This book is dedicated to our dog...



Our super-cute dog, Ella, that sadly passed away. We love you!!

Ella Simon

2002 - 2011



Ella's sister, Billie

Author of Head First Android Development



Jonathan Simon

Before the modern smartphone era, **Jonathan Simon** was coding away at the cool phones of the day, writing low level UI frameworks and debugging tiny screens (back when 176x220 was huge!) with a magnifying glass. Since then, he's worked with all kinds of phones, even the new ones with big fancy schmancy screens.

Before working with mobile devices, Jonathan spent a good six years working on Wall Street designing and building user interfaces for trading systems. And no, it's not his fault the stock market tanked, honest! He also can't give you any stock tips. (Sorry!)

When he's not coding or designing, he's probably hanging out with his wife, Felisa, or their dog, Billie. Otherwise, he's probably riding (or building) a bike or perfecting his espresso extraction.



One of Jonathan's espresso shots. It took MANY of these to write this book.

Table of Contents (Summary)

	Intro	xi
1	Your first app: *Meet Android*	1
2	Give your app an action: *Adding behavior*	41
3	Pictures from space: *Work with feeds*	79
4	When things take time: *Long-running processes*	123
5	Run your app everywhere: *Multiple-device support*	
6	Tablets are not just big phones: *Optimizing for tablets*	
7	Building a list-based app: *Lists and adapters*	167
8	Navigation in Android: *Multi-screen apps*	205
9	Database persistence: *Store your stuff with SQLite*	265
10	RelativeLayout: *It's all relative*	313
11	Give your app some polish: *Tweaking your UI*	345
12	Make the most of what you can use: *Content proficers*	393
i	Leftovers: The *Top Ten Things (We Didn't Cover)*	

Table of Contents (the real thing)

Your First App

So you're thinking: "What makes Android so special?"

Android is a **free and open operating system** from ***Google*** that runs on all kinds of devices from **phones,** to **tablets** and even **televisions**. That's a ton of different devices you can target with *just one platform*! (And the market share is gaining too!) Google provides all of the stuff you need to get started building Android apps **for free**. You can build your Android apps on Macs, Windows, or Unix and publish your apps for next to nothing (with no need for anyone's approval). Ready to get started? Great! You're going to start building your first Android app, but first there are a few things to set up...

1

meet android

Your first app

So you're thinking: "What makes Android so special? "

Android is a **free and open operating system** from ***Google*** that runs on all kinds of devices from **phones,** to **tablets** and even **televisions**. That's a ton of different devices you can target with *just one platform*! (And the market share is gaining too!) Google provides all of the stuff you need to get started building Android apps **for free**. You can build your Android apps on Macs, Windows, or Unix and publish your apps for next to nothing (with no need for anyone's approval). Ready to get started? Great! You're going to start building your first Android app, but first there are a few things to setup...

2

adding behavior

Give your app an action

Apps are interactive! When it comes to apps, it's what your ***users can do*** with your apps that make them love 'em. As you saw in Chapter 1, Android really *separates* out the **visual definition** of your apps (remember all that XML layout and String resource work you just did!) **from the behavior** that's defined in *Java code*. In this chapter, you're going to ***add some behavior*** to the AndroidLove haiku app. And in the process you'll learn how the XML resources and Java work seamlessly together to give you a great way to build your Android apps!

3

work with feeds

Pictures from space!

RSS feeds are everywhere! From weather and stock information to news and blogs, huge amounts of content are distributed in RSS feeds and just waiting to be used in your apps. In fact, the RSS feed publishers want you to use them! In this chapter, you'll learn how to build your own app that incorporates **content** from a public RSS feed on the Web. Along the way, you'll also learn a little more about **layouts**, **permissions,** and **debugging**.

4

long-running processes

When things take time

It would be great if everything happened instantly. Unfortunately, some things just take time. This is especially true on mobile devices, where network latency and the occasionally slow processors in phones can cause things to take a ***bit*** longer. You can make your apps faster with optimizations, but some things just take time. But you ***can*** learn how to **manage long-running processes better**. In this chapter, you'll learn how to show active and passive status to your users. You'll also learn how to perform expensive operations off the UI thread to guarantee your app is always responsive.

5

multiple-device support

Run your app everywhere

There are a lot of different sized Android devices out there. You've got big screens, little screens, and everything in between. And it's your job to support them all! Sounds crazy, right? You're probably thinking right now "***How can I possibly support all of these different devices***?" But with the right strategies, you'll be able to target all of these devices ***in no time*** and with **confidence**. In this chapter, you'll learn how Android classifies all of these different devices into groups based on **screen size** as well as **screen density**. Using these groups, you'll be able to make your app look great on all of these different devices, and all with a **manageable** amount of work!

6

optimizing for tablets

Tablets are not just big phones

Android tablets are coming onto the scene. These new larger-format Android devices give you an entirely new hardware format to present new and cool apps to your users. **But they are not just big phones!** In this chapter, you'll learn hot to get your app up and running on a tablet. You'll learn about the new screen size groupings and also how to use Fragments to combine multiple Activities on a single screen. So more importantly then just running on tablets in this chapter, you'll learn about how to ***make your app work better*** on them.

7

lists and adapters

Building a list-based app

Where would we be without lists? They display read-only information, provide a way for users to select from large data sets, or even act as navigational device by building up an app with a list-based menu structure. In this chapter, you'll learn how to build an app with a list. You learn about where lists store data (in Adapters) and how to customize how that data is rendered in your list. You'll also learn about adding additional layouts to your app (not just the layout that the Wizard creates for you) and turn that into a real view.

8

multi-screen apps

Navigation

Eventually you'll need to build apps with more than one screen.. So far, all of the apps you've built only have a single screen. But the great apps you're going to build may need more than that! In this chapter, you'll learn how to do just that. You'll build an app with a couple of screens, and you'll learn how to create a new Activity and layout which was previously done for you by the Wizard. You'll learn how to navigate between screens and even pass data between them. You'll also learn how to make your own Android context men- the menu that pops up when press the Menu button!

9

database persistence

Store your stuff with SQLite

In memory data storage only gets you so far. In the last chapter, you built a list adapter that only stored data in memory. But if you want the app to ***remember*** data between sessions, you need to **persist** the data. There are a few ways to persist data in Android including writing directly to files and using the built in SQLite database. In this chapter, you'll learn to use the more robust SQLite database solution. You learn how to create and manage your own SQLite database. You'll also learn how to integrate that SQLite datase with the ListView in the TimeTracker app. And don't worry, if you're new to SQL, you'll learn enough to get started and pointers to more information.

10

relativelayout

It's all relative

You've created a few screens now using LinearLayouts (and even nested LinearLayouts). But that will only get you so far. Some of the screens you'll need to build in your own apps will need to do things that you just cant' do with LinearLayout. But don't worry! Android comes with other layouts that you can use. IN this chapter, you'll learn about another super powerful layout called RelativeLayout. This allows you to layout Views on screen relative to each other (hence the name). It's new way to layout your Views, and as you'll see in the chapter, a way to optimize your screen layouts.

11

tweaking your ui

Giving your app some polish

With all the competition in the marketplace, your apps must do more than just work. They have to look great doing it! Sometimes, basic graphics and layouts will work. But other times, you'll need to crank it up a notch. In this chapter, you'll learn about a new layout manager called Relative Layout. It'll let you lay out your screens in ways that you just can't do with LinearLayout and help you code your designs just the way you want them. You'll also learn more techniques for using images to polish up the look and feel of your app. Get your app noticed!

12

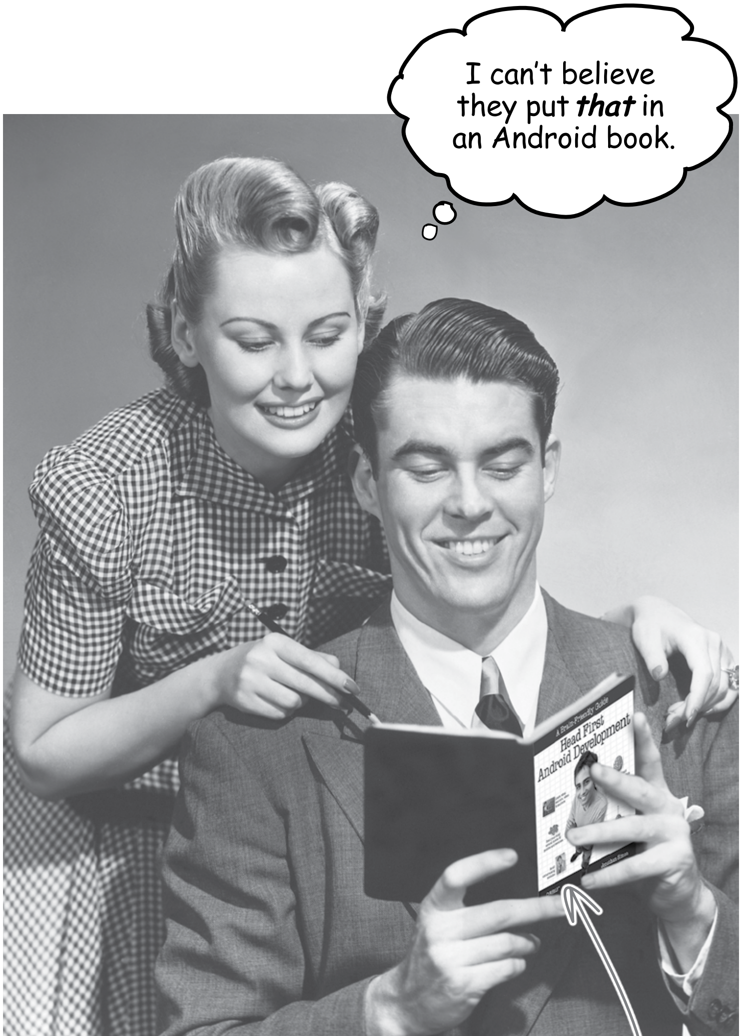
content providers

Make the best of what you can use

You don't want to reinvent the wheel, do you? Of course you don't; you've got apps to build! Well, one of the awesome benefits of Android is the ease in which you can use bits of other applications with content providers. Android apps can expose functionality they want to share and you can use that in your apps. But this doesn't work only for market apps; a number of built-in apps (like the Address Book) expose stuff you can use in your apps too. In this chapter, you'll learn how to use content providers in your app. And who knows, you might like this whole content provider thing so much, you'll decide to provide some of your own content to other apps!

how to use this book

Intro



In this section we answer the burning question: "So why DID they put that in an Android book?"

Who is this book for?

If you can answer "yes" to all of these:

1. Have you done some Java programming, but don't consider yourself a master?
2. Do you want to build mobile apps for an awesome mobile OS that runs on tons of devices?
3. Do you prefer stimulating dinner party conversation to dry, dull, academic lectures?

this book is for you.

Who should probably back away from this book?

If you can answer "yes" to any of these:

1. Have you already mastered Android programming but need a solid reference?
2. Are you solid with the basic Android development fundamentals and are just looking for a guide to its super-advanced features, like ADL or services?
3. Are you afraid to try something different? Would you rather have a root canal than mix stripes with plaid? Do you believe that a technical book can't be serious if it anthropomorphizes control groups and objective functions?

this book is ***not*** for you.



[Note from marketing: this book is for anyone with a credit card.]

We know what you're thinking

"How can *this* be a serious Android development book?"

"What's with all the graphics?"

"Can I actually *learn* it this way?"

We know what your *brain* is thinking



Your brain craves novelty. It's always searching, scanning, *waiting* for something unusual. It was built that way, and it helps you stay alive.

So what does your brain do with all the routine, ordinary, normal things you encounter? Everything it *can* to stop them from interfering with the brain's *real* job—recording things that *matter*. It doesn't bother saving the boring things; they never make it past the "this is obviously not important" filter.

How does your brain *know* what's important? Suppose you're out for a day hike and a tiger jumps in front of you, what happens inside your head and body?

Neurons fire. Emotions crank up. *Chemicals surge*.

And that's how your brain knows...

This must be important! Don't forget it!

But imagine you're at home, or in a library. It's a safe, warm, tiger-free zone. You're studying. Getting ready for an exam. Or trying to learn some tough technical topic your boss thinks will take a week, ten days at the most.



Just one problem. Your brain's trying to do you a big favor. It's trying to make sure that this *obviously* non-important content doesn't clutter up scarce resources. Resources that are better spent storing the really *big* things. Like tigers. Like the danger of fire. Like how you should never have posted those "party" photos on your Facebook page. And there's no simple way to tell your brain, "Hey brain, thank you very much, but no matter how dull this book is, and how little I'm registering on the emotional Richter scale right now, I really *do* want you to keep this stuff around."

We think of a "Head First" reader as a learner.

So what does it take to *learn* something? First, you have to *get* it, then make sure you don't *forget* it. It's not about pushing facts into your head. Based on the latest research in cognitive science, neurobiology, and educational psychology, *learning* takes a lot more than text on a page. We know what turns your brain on.

Some of the Head First learning principles:



Make it visual. Images are far more memorable than words alone, and make learning much more effective (up to 89% improvement in recall and transfer studies). It also makes things more understandable. **Put the words within or near the graphics** they relate to, rather than on the bottom or on another page, and learners will be up to *twice* as likely to solve problems related to the content.

Use a conversational and personalized style. In recent studies, students performed up to 40% better on post-learning tests if the content spoke directly to the reader, using a first-person, conversational style rather than taking a formal tone. Tell stories instead of lecturing. Use casual language. Don't take yourself too seriously. Which would *you* pay more attention to: a stimulating dinner party companion, or a lecture?



Get the learner to think more deeply. In other words, unless you actively flex your neurons, nothing much happens in your head. A reader has to be motivated, engaged, curious, and inspired to solve problems, draw conclusions, and generate new knowledge. And for that, you need challenges, exercises, and thought-provoking questions, and activities that involve both sides of the brain and multiple senses.

Get—and keep—the reader's attention. We've all had the "I really want to learn this but I can't stay awake past page one" experience. Your brain pays attention to things that are out of the ordinary, interesting, strange, eye-catching, unexpected. Learning a new, tough, technical topic doesn't have to be boring. Your brain will learn much more quickly if it's not.



Touch their emotions. We now know that your ability to remember something is largely dependent on its emotional content. You remember what you care about. You remember when you *feel* something. No, we're not talking heart-wrenching stories about a boy and his dog. We're talking emotions like surprise, curiosity, fun, "what the...?" , and the feeling of "I Rule!" that comes when you solve a puzzle, learn something everybody else thinks is hard, or realize you know something that "I'm more technical than thou" Bob from engineering *doesn't*.

Metacognition: thinking about thinking



If you really want to learn, and you want to learn more quickly and more deeply, pay attention to how you pay attention. Think about how you think. Learn how you learn.

Most of us did not take courses on metacognition or learning theory when we were growing up. We were *expected* to learn, but rarely *taught* to learn.

But we assume that if you're holding this book, you really want to learn Android. And you probably don't want to spend a lot of time. If you want to use what you read in this book, you need to *remember* what you read. And for that, you've got to *understand* it. To get the most from this book, or *any* book or learning experience, take responsibility for your brain. Your brain on *this* content.

The trick is to get your brain to see the new material you're learning as Really Important. Crucial to your well-being. As important as a tiger. Otherwise, you're in for a constant battle, with your brain doing its best to keep the new content from sticking.

So just how *DO* you get your brain to treat Android like it was a hungry tiger?

There's the slow, tedious way, or the faster, more effective way. The slow way is about sheer repetition. You obviously know that you *are* able to learn and remember even the dullest of topics if you keep pounding the same thing into your brain. With enough repetition, your brain says, "This doesn't *feel* important to him, but he keeps looking at the same thing *over* and *over* and *over*, so I suppose it must be."

The faster way is to do ***anything that increases brain activity,*** especially different *types* of brain activity. The things on the previous page are a big part of the solution, and they're all things that have been proven to help your brain work in your favor. For example, studies show that putting words *within* the pictures they describe (as opposed to somewhere else in the page, like a caption or in the body text) causes your brain to try to makes sense of how the words and picture relate, and this causes more neurons to fire. More neurons firing = more chances for your brain to *get* that this is something worth paying attention to, and possibly recording.

A conversational style helps because people tend to pay more attention when they perceive that they're in a conversation, since they're expected to follow along and hold up their end. The amazing thing is, your brain doesn't necessarily *care* that the "conversation" is between you and a book! On the other hand, if the writing style is formal and dry, your brain perceives it the same way you experience being lectured to while sitting in a roomful of passive attendees. No need to stay awake.

But pictures and conversational style are just the beginning…

Here's what WE did:

We used ***pictures***, because your brain is tuned for visuals, not text. As far as your brain's concerned, a picture really *is* worth a thousand words. And when text and pictures work together, we embedded the text *in* the pictures because your brain works more effectively when the text is *within* the thing the text refers to, as opposed to in a caption or buried in the text somewhere.

We used ***redundancy***, saying the same thing in *different* ways and with different media types, and *multiple senses*, to increase the chance that the content gets coded into more than one area of your brain.

We used concepts and pictures in ***unexpected*** ways because your brain is tuned for novelty, and we used pictures and ideas with at least *some* ***emotional*** *content*, because your brain is tuned to pay attention to the biochemistry of emotions. That which causes you to *feel* something is more likely to be remembered, even if that feeling is nothing more than a little ***humor***, ***surprise***, or ***interest.***

We used a personalized, ***conversational style***, because your brain is tuned to pay more attention when it believes you're in a conversation than if it thinks you're passively listening to a presentation. Your brain does this even when you're *reading*.

We included more than 80 ***activities***, because your brain is tuned to learn and remember more when you ***do*** things than when you *read* about things. And we made the exercises challenging-yet-do-able, because that's what most people prefer.

We used ***multiple learning styles***, because *you* might prefer step-by-step procedures, while someone else wants to understand the big picture first, and someone else just wants to see an example. But regardless of your own learning preference, *everyone* benefits from seeing the same content represented in multiple ways.

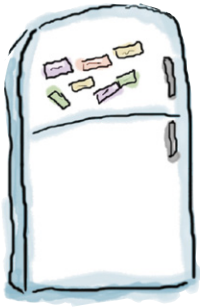
We include content for ***both sides of your brain***, because the more of your brain you engage, the more likely you are to learn and remember, and the longer you can stay focused. Since working one side of the brain often means giving the other side a chance to rest, you can be more productive at learning for a longer period of time.

And we included ***stories*** and exercises that present ***more than one point of view,*** because your brain is tuned to learn more deeply when it's forced to make evaluations and judgments.

We included ***challenges***, with exercises, and by asking ***questions*** that don't always have a straight answer, because your brain is tuned to learn and remember when it has to *work* at something. Think about it—you can't get your *body* in shape just by *watching* people at the gym. But we did our best to make sure that when you're working hard, it's on the *right* things. That ***you're not spending one extra dendrite*** processing a hard-to-understand example, or parsing difficult, jargon-laden, or overly terse text.

We used ***people***. In stories, examples, pictures, etc., because, well, because *you're* a person. And your brain pays more attention to *people* than it does to *things*.





Here's what YOU can do to bend your brain into submission

So, we did our part. The rest is up to you. These tips are a starting point; listen to your brain and figure out what works for you and what doesn't. Try new things.

1. **Slow down. The more you understand, the less you have to memorize.**
   Don't just *read*. Stop and think. When the book asks you a question, don't just skip to the answer. Imagine that someone really *is* asking the question. The more deeply you force your brain to think, the better chance you have of learning and remembering.

2. **Do the exercises. Write your own notes.**
   We put them in, but if we did them for you, that would be like having someone else do your workouts for you. And don't just *look* at the exercises. **Use a pencil.** There's plenty of evidence that physical activity *while* learning can increase the learning.

3. **Read the "There are No Dumb Questions"**
   That means all of them. They're not optional sidebars, ***they're part of the core content!*** Don't skip them.

4. **Make this the last thing you read before bed. Or at least the last challenging thing.**
   Part of the learning (especially the transfer to long-term memory) happens *after* you put the book down. Your brain needs time on its own, to do more processing. If you put in something new during that processing time, some of what you just learned will be lost.

5. **Talk about it. Out loud.**
   Speaking activates a different part of the brain. If you're trying to understand something, or increase your chance of remembering it later, say it out loud. Better still, try to explain it out loud to someone else. You'll learn more quickly, and you might uncover ideas you hadn't known were there when you were reading about it.

6. **Drink water. Lots of it.**
   Your brain works best in a nice bath of fluid. Dehydration (which can happen before you ever feel thirsty) decreases cognitive function.

7. **Listen to your brain.**
   Pay attention to whether your brain is getting overloaded. If you find yourself starting to skim the surface or forget what you just read, it's time for a break. Once you go past a certain point, you won't learn faster by trying to shove more in, and you might even hurt the process.

8. **Feel something.**
   Your brain needs to know that this *matters*. Get involved with the stories. Make up your own captions for the photos. Groaning over a bad joke is *still* better than feeling nothing at all.

9. **Get your hands dirty!**
   There's only one way to learn to Android: get your hands dirty. And that's what you're going to do throughout this book. Android Development is a skill, and the only way to get good at it is to practice. We're going to give you a lot of practice: every chapter has exercises that pose a problem for you to solve. Don't just skip over them—a lot of the learning happens when you solve the exercises. We included a solution to each exercise—don't be afraid to peek at the solution if you get stuck! (It's easy to get snagged on something small.) But try to solve the problem before you look at the solution. And definitely get it working before you move on to the next part of the book.

The technical review team

Technical Reviewers:

Paul Barry

David Griffith

Frank Maker

Herve Guihot

Acknowledgments

My editor:

Brian Sawyer kept the ball rolling all through this process. I had to learn a lot to pull this off, and he always made sure I was hooked up with the right folks to help me ***get it done***!

My design editor:

Dawn Griffiths used her keen design sense and Head First touch to make these pages more beautiful and more learner friendly.

My wife:

As with everything else in my life, this book would not have been possible without my totally super awesome wife, **Felisa**! She listened to countless hours of discussion on Android, as well as the finer points of teaching it Head First. Undoubtedly, she rocks!



Brian Sawyer

Felisa Wolfe-Simon



Safari® Books Online

When you see a Safari® icon on the cover of your favorite technology book that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at `http://my.safaribooksonline.com/?portal=oreilly.`

1 meet android

Your first app



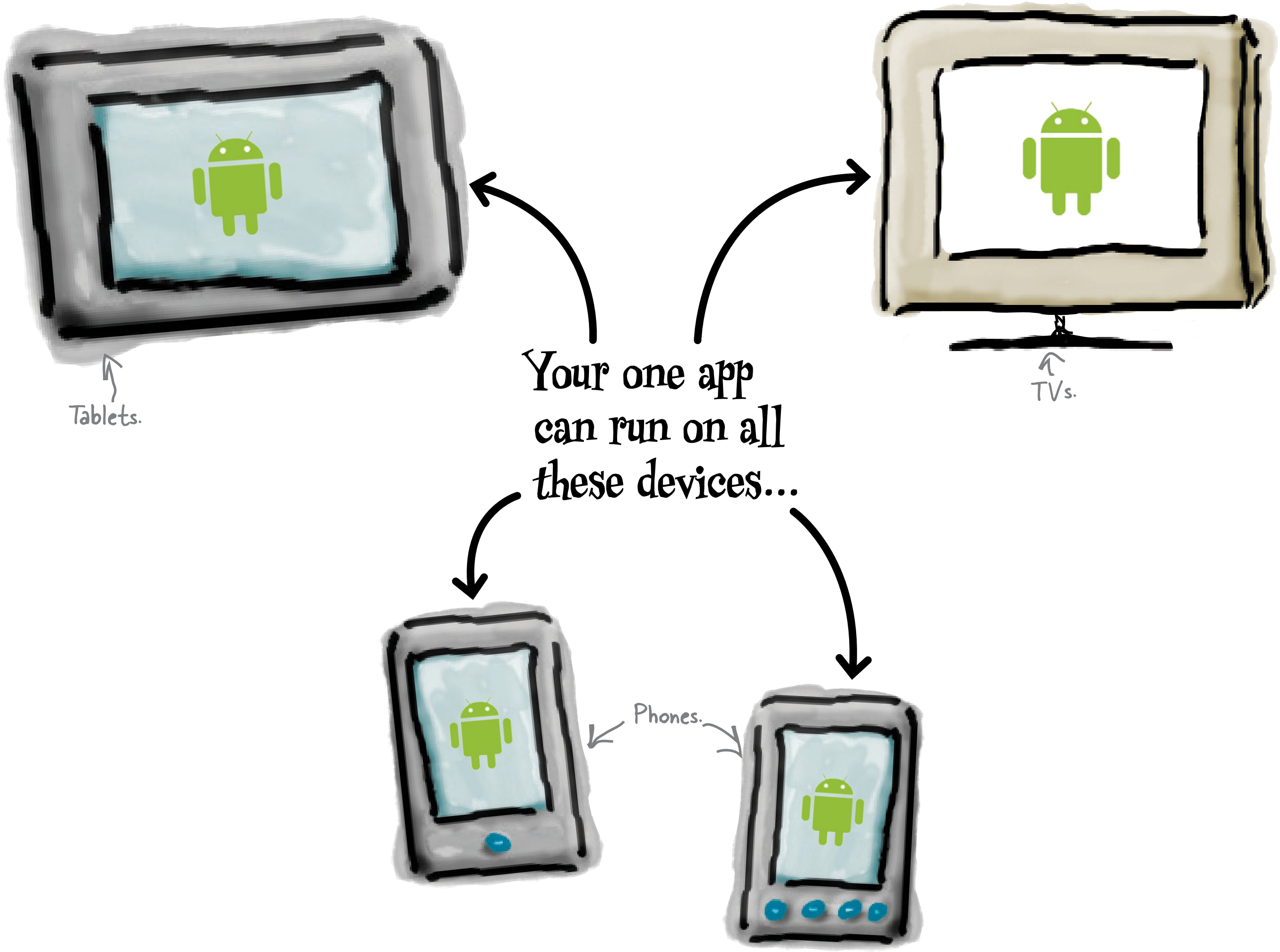
So you're thinking: "What makes Android so special? " Android is a **free and open operating system** from ***Google*** that runs on all kinds of devices from **phones,** to **tablets** and even **televisions**. That's a ton of different devices you can target with *just one platform*. (And the market share is gaining too). Google provides everything you need to get started building Android apps *for free*. And you can build your Android apps on either Mac, Windows, or Unix and publish your apps for next to nothing (and with no need for anyone's approval). Ready to get started? Great! You're going to start building your first Android app, but first there are a few things to setup...

So you want to build an Android app...

Maybe your an Android user, you already know Java and want to get in on the mobile craze, or you just love the open operating system and hardware distribution choices of Android. Whatever your reason, you've come to the right place.

Android already runs on a TON of different devices!

With careful planning, you're app can run on all of these Android powered devices. From phones and tablets, to TVs and even home automation, Android is spreading quickly.



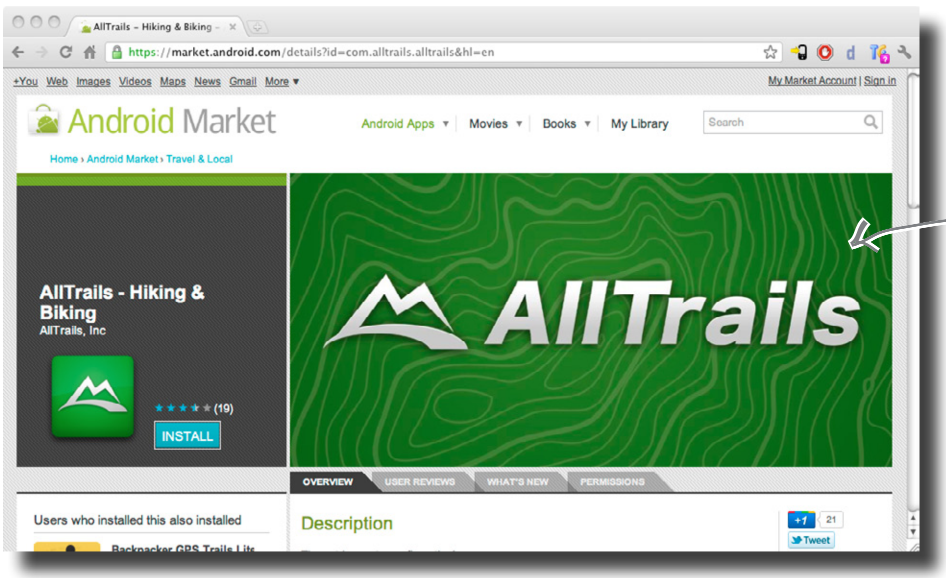
And it's growing!

That's a LOT of devices in one day!

> **"Over 500,000 Android devices [are] activated every day"**
>
> -- Google's Head of Android, Andy Rubin, via Twitter

Just check out the Android Market

The Android Market has a ton of apps. There are or course games (because we all love playing games on our phones), but also really great apps that just **make our lives better** like navigation and commuting schedule apps.



The Android Market web view for an outdoor exploration app AllTrails.

There are a lot of mobile platforms out there, but with Android's presence and growth, **everyone** is building out their Android apps. Welcome to Android, it's a great place to be!

Before you dig into your first app, let's take a look at exactly *what Android is* and who's responsible for it...

So tell me about Android...

Android is a **mobile operating syetem**, but it's a lot more than that too. There is a whole **ecosystem**, a complete platform, and community that supports Android apps getting built and on to new Android based hardware devices.

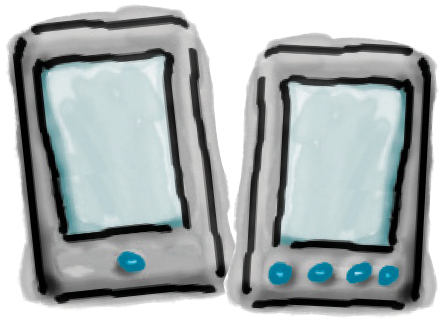
1 Google maintains Android

Google maintains Android, but it's free to use. Device manufacturers and carriers can modify me, and developers can build apps for free.



2 Hardware manufacturers build a device

Hardware manufactures can use the Android operating system and build special hardware around it. Manufacturers can even modify Android to implement custom functionality for their devices.



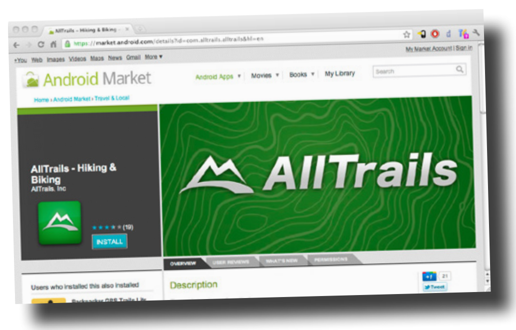
3 Google gives you the tools

Google freely distributes the tools for you to build your own Android apps. And you can build your apps on multiple platforms: Mac, Windows, Linux...

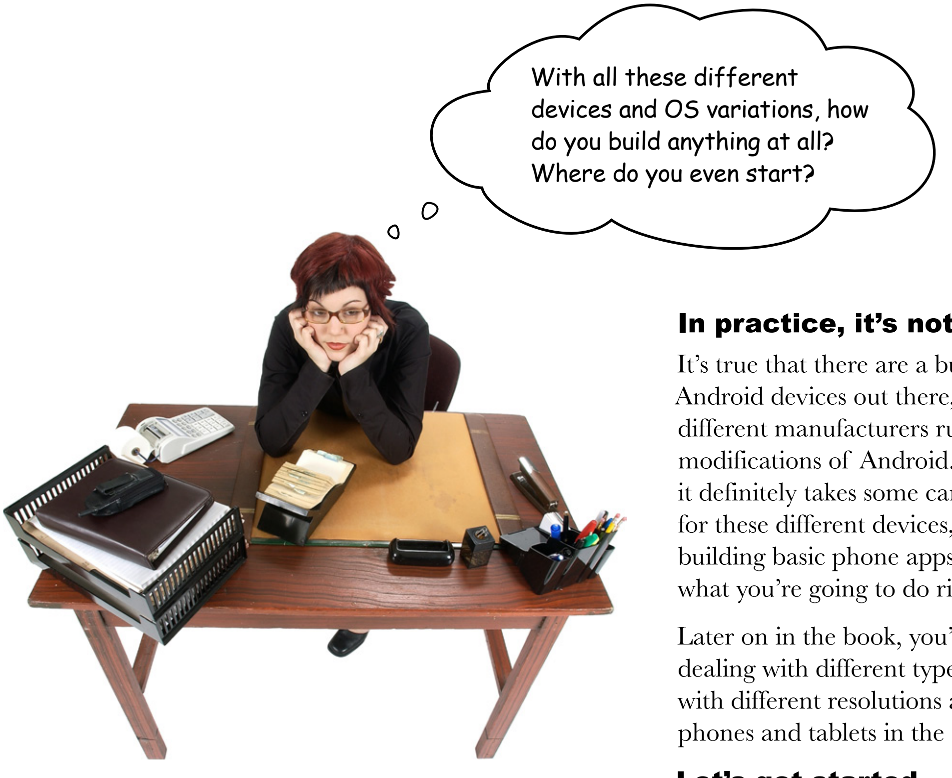


4 Google also runs a Market

This is where your users can download their apps right to their phones. Google runs one market, but there are also others run by Amazon, and Verizon for example. But the biggest one is still Google's.



Are you ready to get started?



In practice, it's not so bad!

It's true that there are a bunch of different Android devices out there, from all kinds of different manufacturers running different modifications of Android. *Sounds crazy right?* While it definitely takes some care tuning your apps for these different devices, you can get started building basic phone apps *really easily*. And that's what you're going to do right now.

Later on in the book, you'll learn strategies for dealing with different types of devices like phones with different resolutions and even designing for phones and tablets in the same app.

Let's get started.

Meet Pajama Death

It's time to introduce you to an awesome rock duo called the **Pajama Death**! They love Android and love to sing about it!



They write all of their song lyrics in the form of a haiku

A **haiku** is an ancient Japanese form of poetry. Each poem consists of 3 lines - the first line having 5 syllables, the second 7 syllables, and the third line 5 syllables just like the first. These poems are meant to be meaningful, yet compact... just like your Android apps!

They're about to play their favorite song for you! This one's called... Android Love!



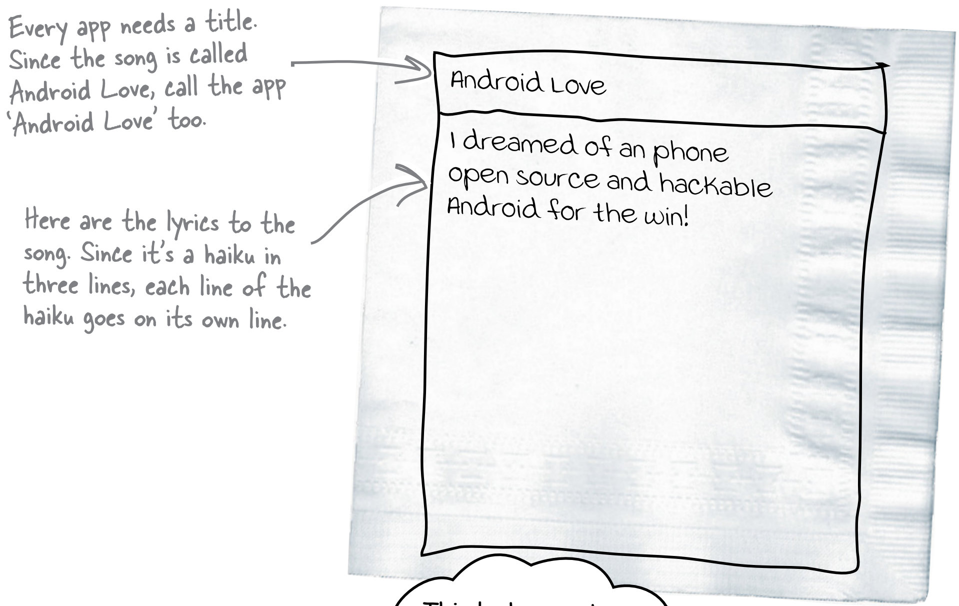
But they need your help!

They want to make an app with the **Android Love** lyrics to hand out to their fans. But they are Android ***users*** not Android ***developers***. They heard that you were learning to build your own Android apps. They were wondering if you would build the app for them. And how could you say no? Of course you'll do it, you're a huge fan!

OK, let's get started...

Getting started

Just asking you to build an app isn't a lot to go on. So the ***Pajama Death*** made a napkin sketch of what they want the app to look like. It's an app showing the haiku, with each line of the haiku on a new line.





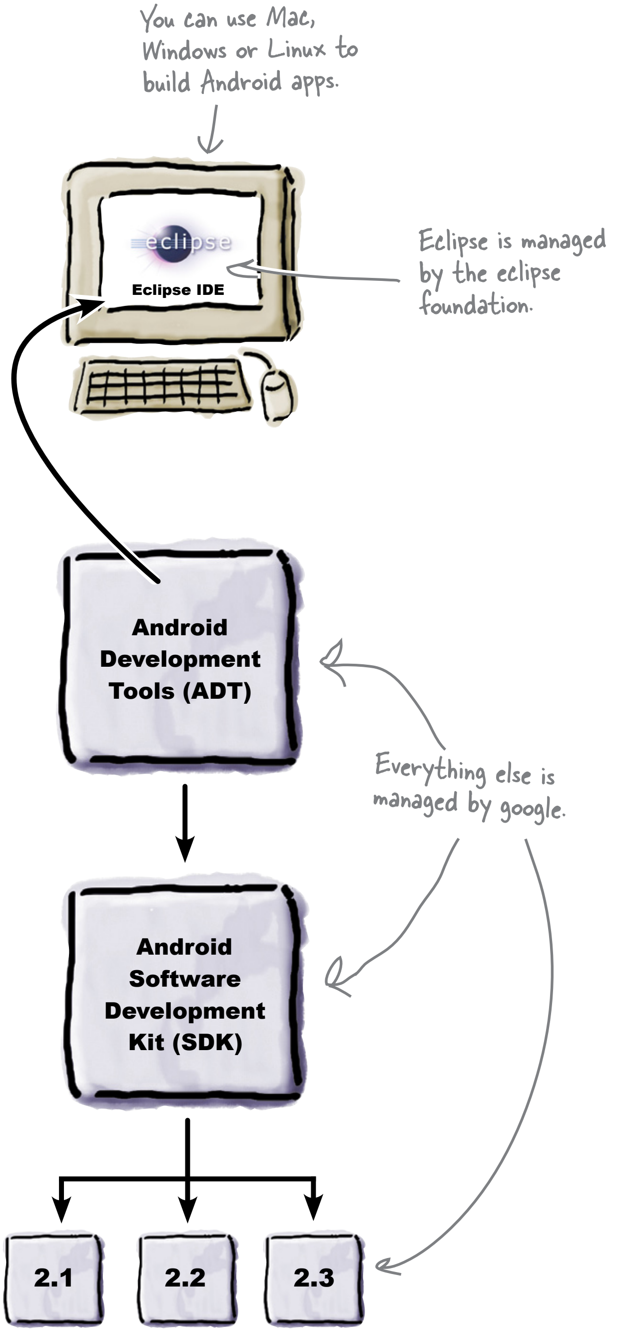


First you've got some setup to do

Since this is your first Android app, you'll need to setup your development environment. Let's start with a quick look at what you need in your development environment to build Android apps. Form there, you'll install your own development environment, then build the app for ***Pajama Death***!

Meet the android development environment

The Android development environment is made up of *several* parts that seamlessly work together for you to build Android apps. Let's take a closer look at each one.



1. **Eclipse Integrated Development Enviroment (IDE)**
   The Eclipse Integrated Development Environment (IDE for short) is where you'll write your code. Eclipse is a generic IDE, not specific to Android development. It's managed by the Eclipse foundation.

2. **Android Development Tools (ADT)**
   The Android Development Tools (ADT) is an Eclipse plugin that adds Android specific functionality to Eclipse.

3. **Software Development Kit (SDK)**
   The Android Software Development Kit (SDK) contains all of the lower level tools to build, run and test your Android apps. The ADT is really just a user interface, and the guts of the app building all happens here in the ADT.

4. **Android Packages**
   You can develop and support multiple versions of Android from the same developmentw environment. These packages add functionality to the base SDK to let you develop for that Android

Choosing your IDE

Eclipse may be a fine IDE, but what if you don't want to use it. You may have your own IDE of choice that you'd rather use...



You don't *have* to use Eclipse.

But it certainly makes things ***easier***. The full integrated Android development environment works well as a whole to help you easily build Android apps.

But everything you need to build and test your Android apps is the **Android SDK and Android Packages**. If you really cant live without your favorite development environment,. you can use it in conjunction with the SDK without Eclipse and still build Android apps.

Even though you can use the SDK without Eclipse, all of the examples in this book will use Eclipse and the ADT plugin.



Set up your development environment

You won't be able to build your apps until your development environment is setup! Follow our nifty Android development environment setup instructions over the next few pages and you'll be ready to build your apps!

Turn the page for instructions on setting up your own Android development environment...

Download, install and launch eclipse

Eclipse is a free and open source IDE managed by the Eclipse foundation (started and managed by IBM, but a very open community). You can download Eclipse for free from the **eclipse.org**. There are a number of different versions of Eclipse optimized for different types of development. You should download the latest version of **Eclipse Classic** for your Operating System.

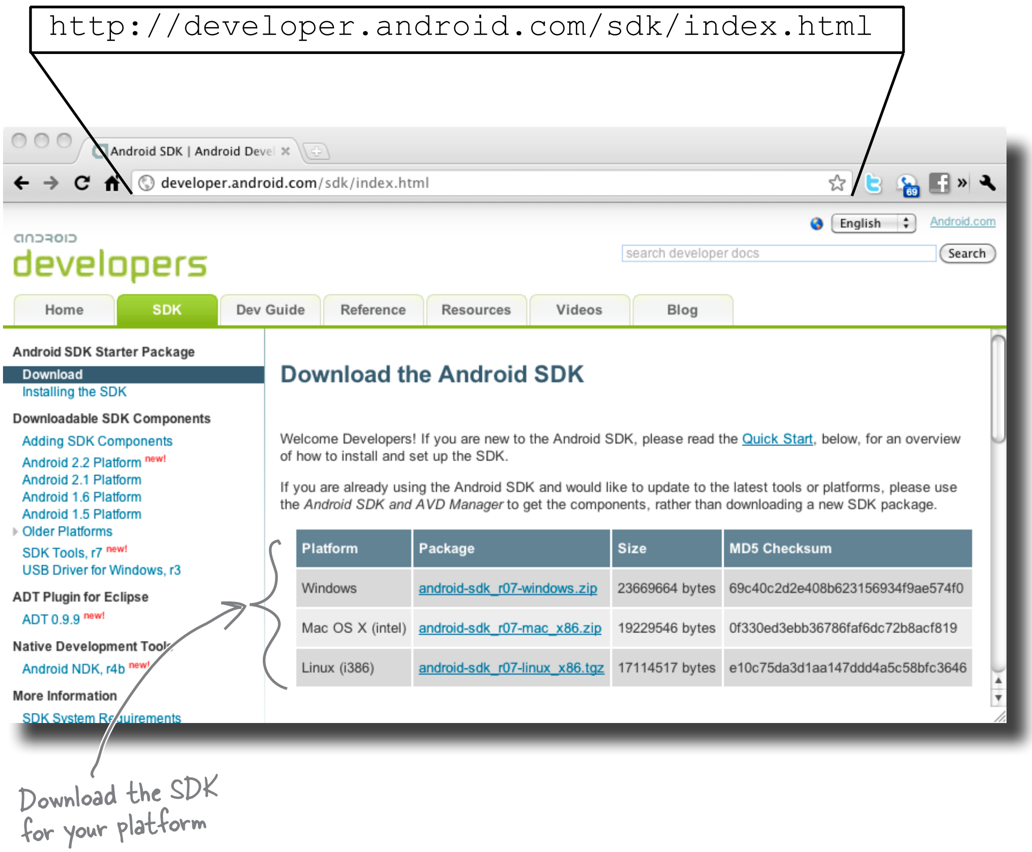


After you download Eclipse, follow the installation instructions for your platform and launch Eclipse. When you launch Eclipse for the first time, you will be prompted to enter a ***workspace location***; a directory where all of your Eclipse projects and settings will be stored. Feel free to use the default or enter your own.



Doanload and install the SDK

The Android SDK contains the core tools needed to build and run Android apps. This includes the Android emulator, builder, docs and more. You can download the SDK from android.developer.com.



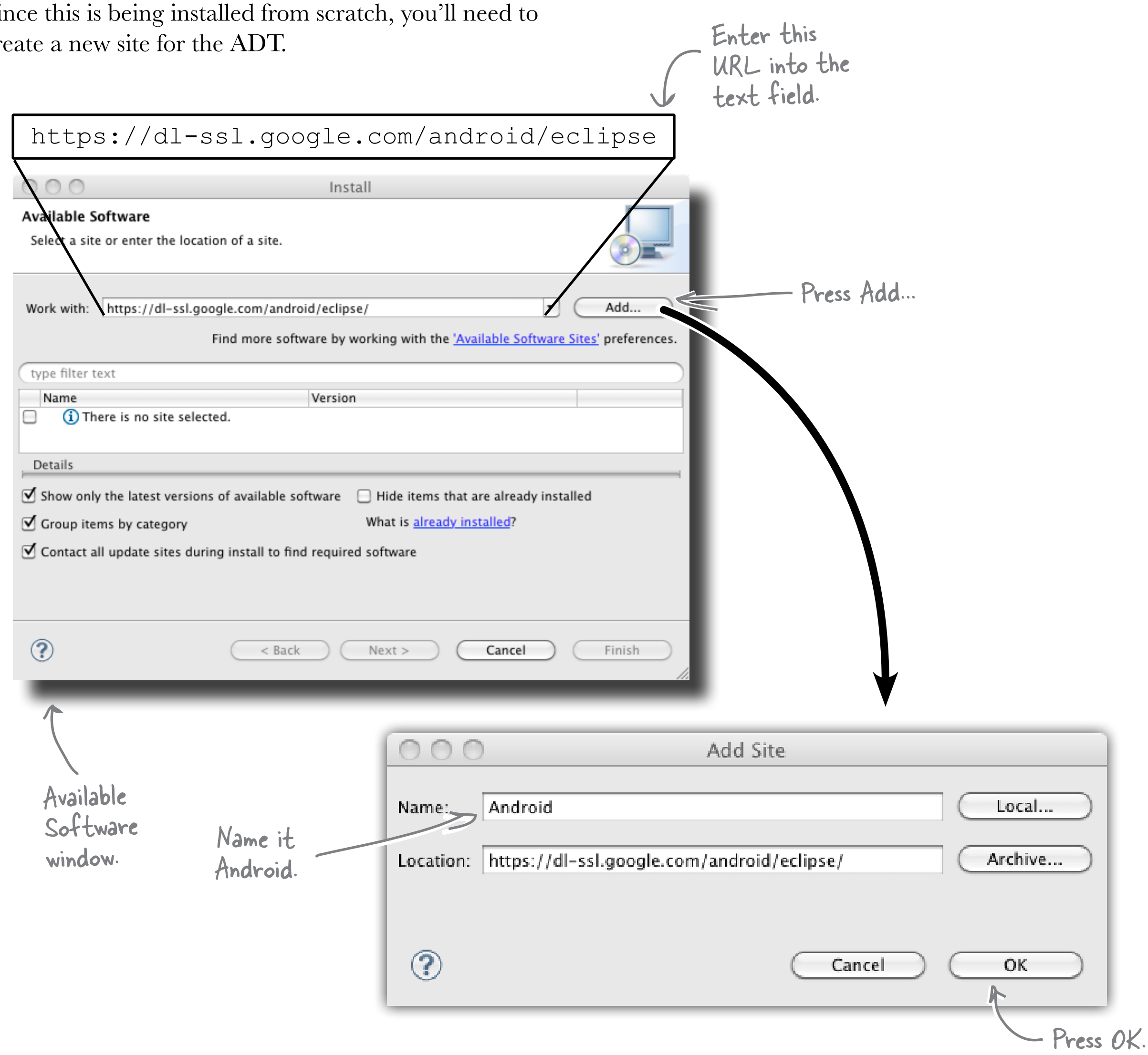
Once you download the SDK zip file, unzip it to your hard drive and the SDK is ready to go.

Now let's setup the ADT...

Install the ADT

The Android Development Tools (ADT) are the glue that seamlessly connects the Android specific SDK with Eclipse. The ADT is an Eclipse plugin, and it installs through the standard Eclipse plugin installation mechanism (so this should look very familiar if you're an experienced Eclipse user).

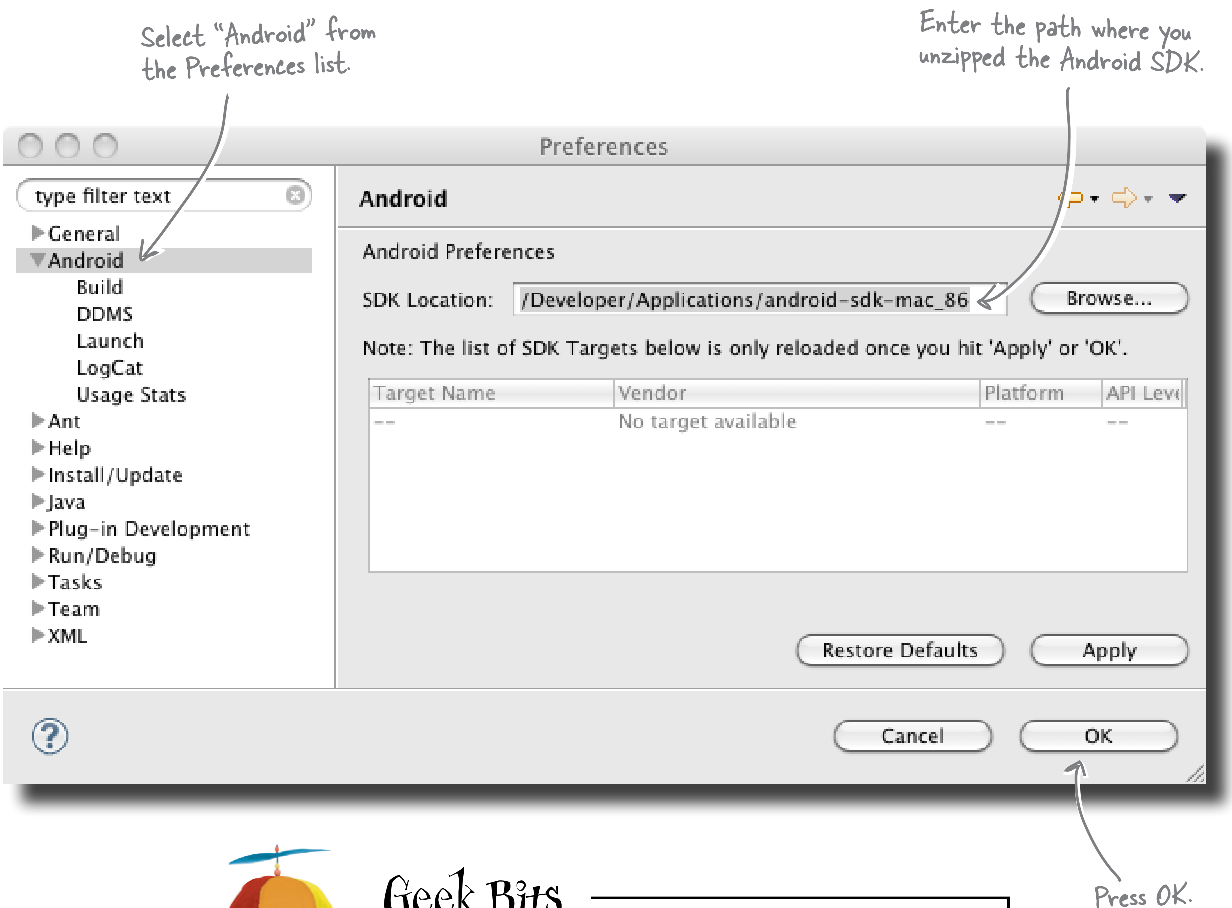
From your Eclipse window, select Help → Install new software. This will bring up the Available Software window. Since this is being installed from scratch, you'll need to create a new site for the ADT.



Configure the ADT

The ADT is just the glue between the SDK and Eclipse, so the ADT needs to know where the SDK is installed.

Set the SDK location in the ADT by going to Window → Preferences in Eclipse, selecting Android from the left panel, and selecting the directory where you installed the Android SDK.





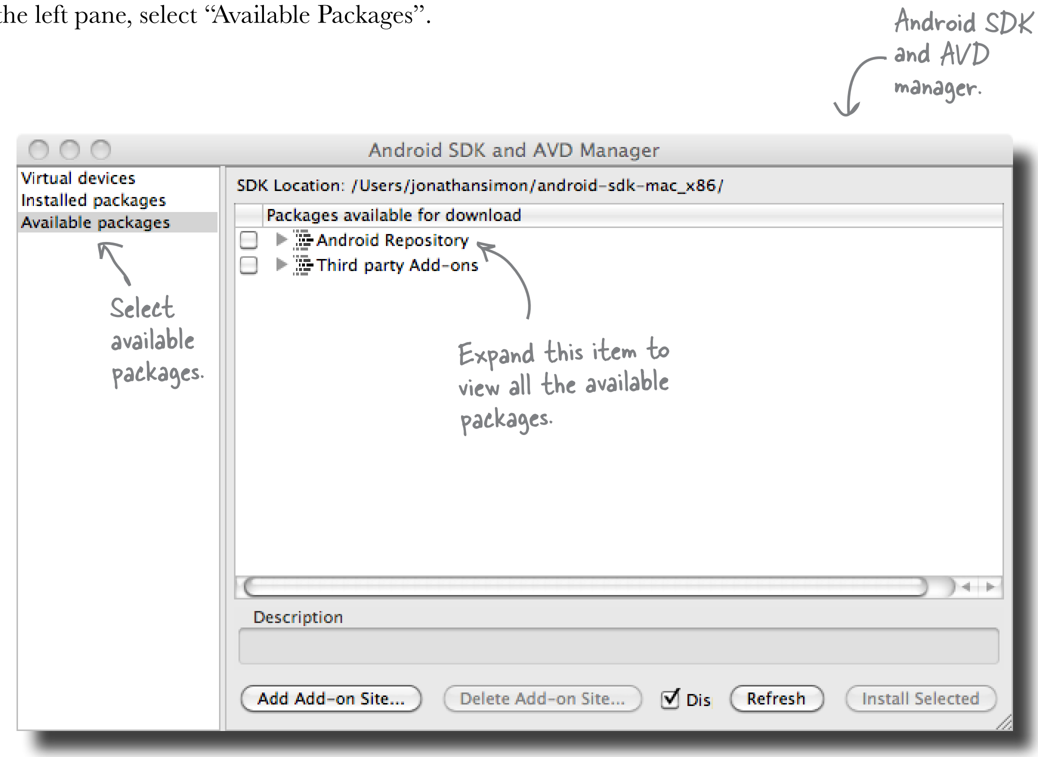
Geek Bits

It's a good idea to add the `<SDK-install-directory>/tools` directory to your path. The SDK includes a number of command line tools and it's convenient to be able to launch them without having to type in complete paths.

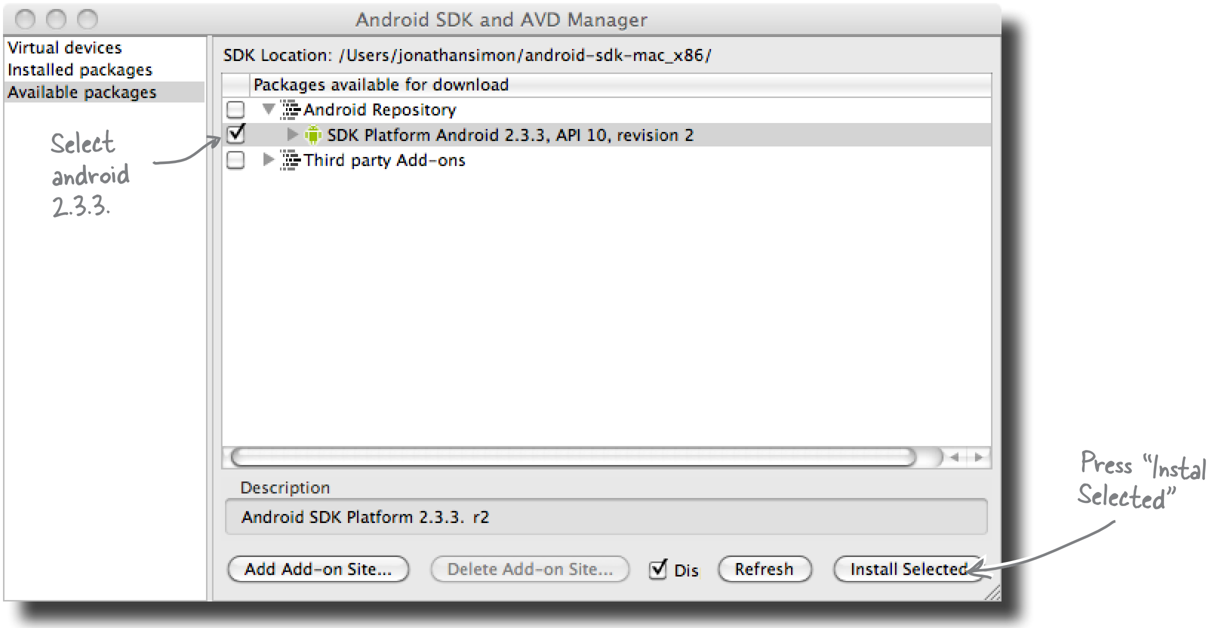
Install android packages

The SDK is designed to allow you to work with multiple versions of Android in the same development environment. To keep downloads small, the SDK version packages are separated from the SDK. (This also allows you to update to new versions of Android without having to redownload the entire SDK. ***Pretty slick!***)

You can configure the installed packages in the SDK from the Android SDK and AVD Manager (another added bonus of the ADT). Open the manager by selecting Window → Android SDK and AVD Manager. From the left pane, select "Available Packages".



When you expand the tree node, you'll see a combination of SDK Tools, SDK platforms, samples documentation and more. These are all plugins to the SDK that you can add to expand the functionality of the SDK. (This way you can download and install the SDK once and keep adding new functionality to it as new versions come out).



Do this! → Select "SDK Platform Android 2.3.3" and press "Install Selected".

there are no Dumb Questions

Q: What about the samples should I install those?

A: Google put together a set of sample apps that show off a bunch of features and techniques in the platform. They won't be used in the book, but they are extremely useful. If you want to learn about something not covered in the book, the samples are a great place to start.

Q: And what about Tools? Should I install those too?

A: The tools inside the SDK can also get updated as new functionality is released in the Android platform. It's a good idea to keep these up to date.

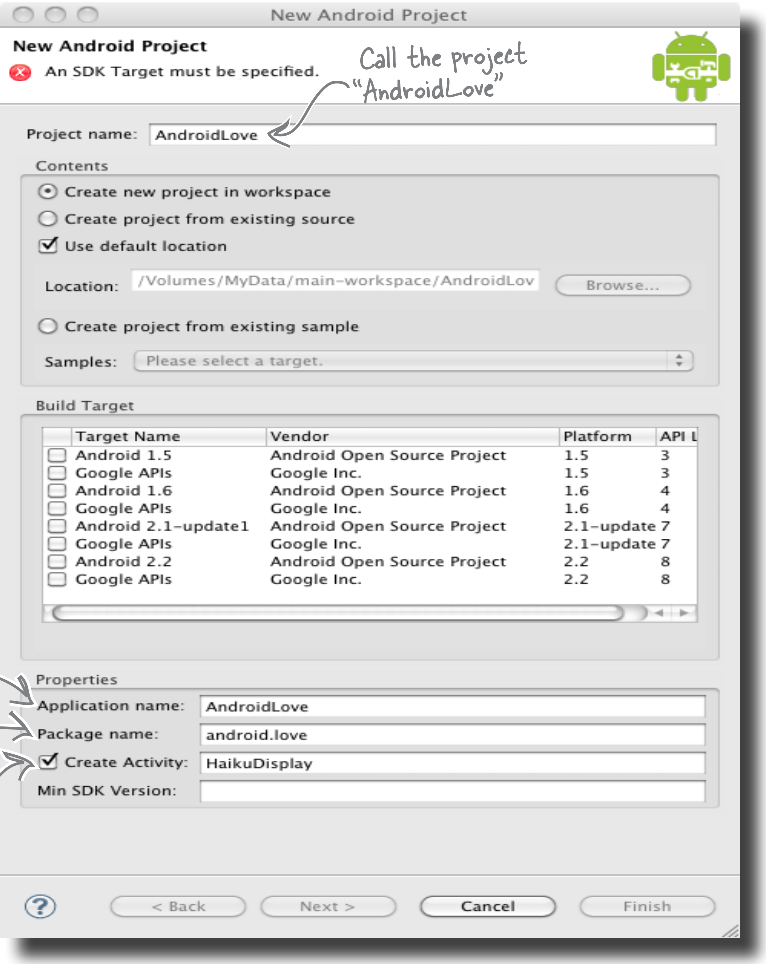
Make a new Android app project

Now that you have your environment setup, it's time to make your first project.

The Eclipse ADT plugin comes with a Wizard to create new Android apps. All you have to do is enter a few bits of information into the wizard, and it makes a fully functional (but very ***boring***) application for you.

Launch the New Android Project wizard by going to File → New → Android Project, then fill in the fields to make your new project!

New Android Project wizard.



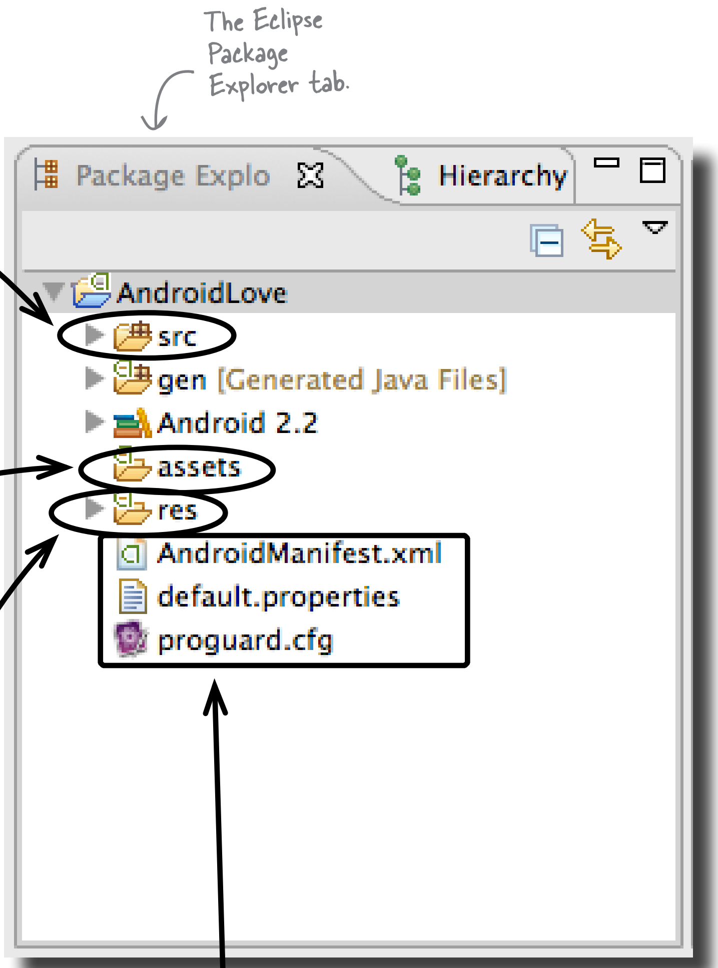
Call the project "AndroidLove". This is the app name your users will see.

Set the package name to "android.love". This will be used for the java package name in your project.

Leave "Create Activity" checked Call the Activity "HaikuDisplay". This will generate the behavior code for your screen displaying the hauki.

What's in an Android project?

Wizards are great because they do a lot of basic setup for you. But what did that wizard do anyway? Here's a quick look at the basic Android project that the wizard created. To look at the project contents, click on the "***Package Explorer***" tab in Eclipse.



App Behavior in Java code

The behavior of Android apps is built with Java code. This code controls what happens when buttons are pressed, calls to servers, and any other behavior that your app is doing. Your android projects have a source directory where all of the Java code lives.

Binary assets

Great apps need to do more than just deliver great functionality... they need to look great doing it. You'll be using images to style your app and give them custom polished looks. The images and other raw binary resources in this directory are included in your app.

Resources and XML layouts

For Android apps, layouts are primarily defined in XML rather than code. All sorts of other properties are defined in XML too - like string values, colors, and more. These XML files are stored in the res directory.

Configuration files

Your app now has Java code, XML resources, and binary assets that define it. Configuration files are the glue that holds all of it together. Everything from the title of your app on the Android home screen, to the different screens in your app are defined in these configuration files.

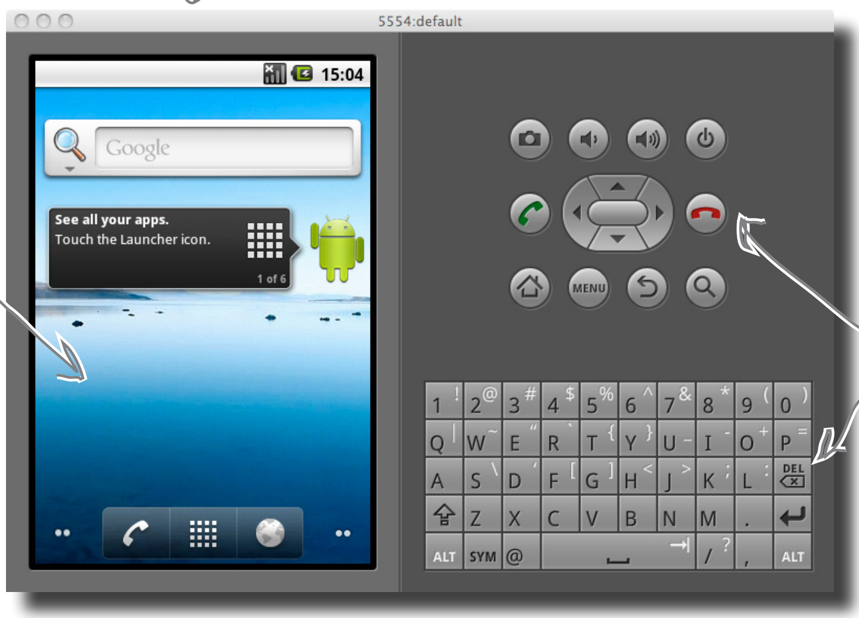
Run the project!

At this point, your new project is all ready to run! The wizard not only setup a project for you, but also created a very basic runnable Android app. **How *cool* is that!**

Test run your apps using the Android emulator

The Android SDK includes an Android emulator desktop application that simulates a complete running Android device. It runs a full basic android operating system and the default set of Android apps. It's obviously not a complete hardware Android device, but it's about as close as you can get with hardware emulation!

Running Android emulator.



You can simulate touch screen "presses" by clicking on the screen with your mouse.

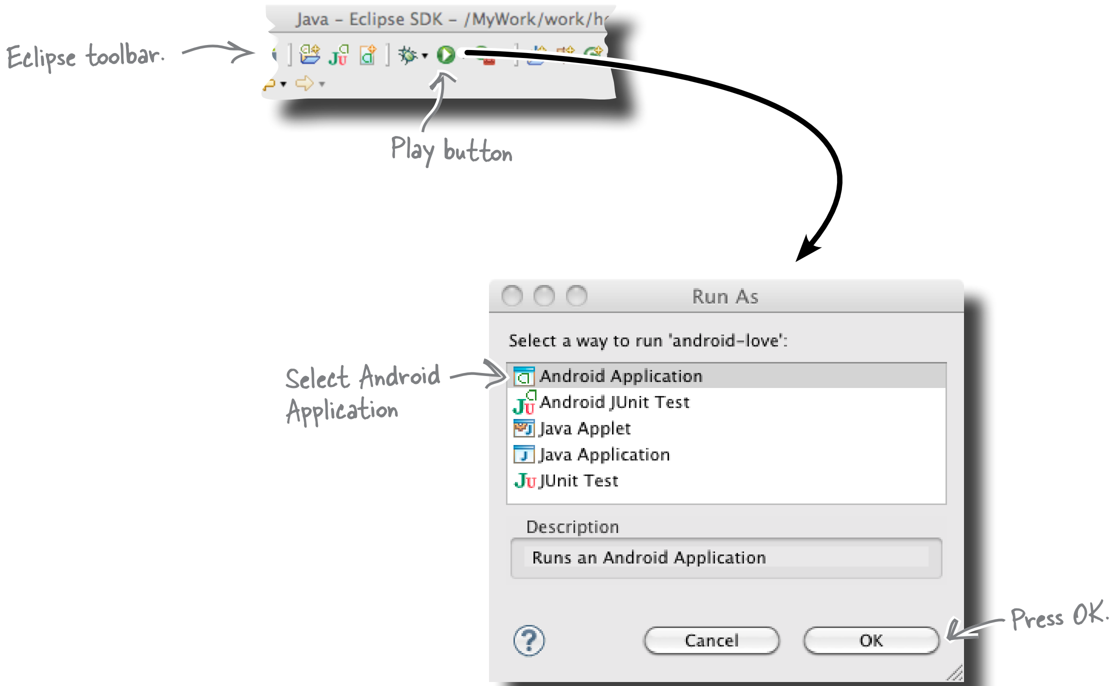
The emulator also includes hardware buttons like a keyboard and the Android hard buttons.



Test Drive

To run an Android app from Eclipse, select "Run → Run" and you'll see a dialog that prompts you for how you want to run the project. Since your project is an Android app, select "Android Application" and click on "OK".

Alternatively, you can run your android apps by pressing the "play" button on the Eclipse toolbar.



But instead of seeing an Android app running, you'll see the following dialog.



Why won't the app run?

The app didn't run, and instead you were faced with a dialog with an error about a target not being found and asking you to create a Virtual Device.

Wait, I thought you said I could run the app right out of the box! Were you lying to me?



The app *is* fine to run.

The issue isn't with the app the wizard generated, the issue is that there no way to run it. Your Android development environment can built apps for multiple Android versions, hardware configurations and screen sizes. So when you try and run your app, the Android tools don't know what type of device you want to run your app on.

The solution is to create Android Virtual Devices (or AVD for short) that defines a particular device's software version and hardware format to run your app in. You can think of an AVD as like a saved emulator configuration.

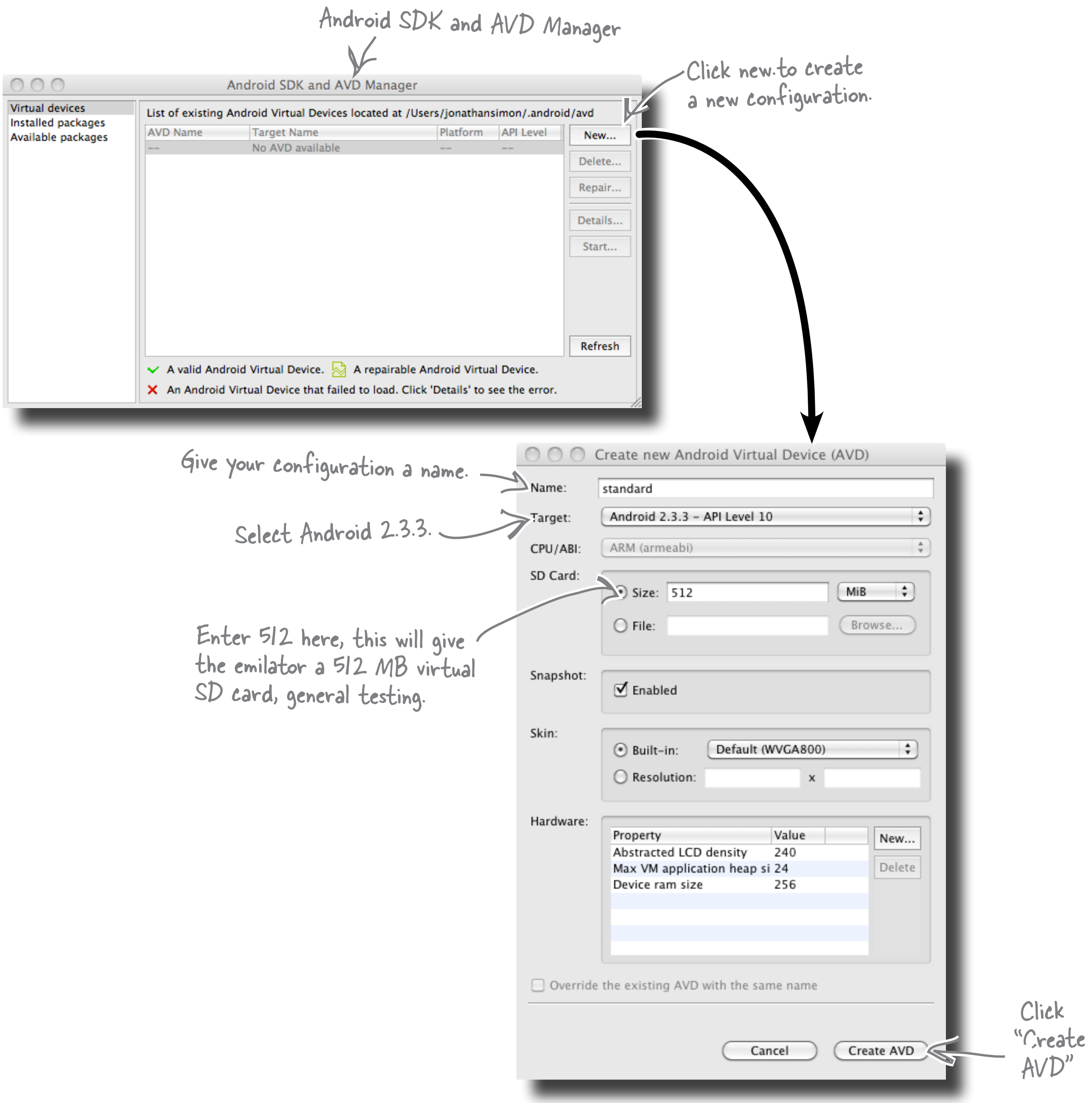
Since you don't have an AVD setup already (and there are no stock AVDs in the Android SDK) you have to make your own.

Do this!

Click Yes on the dialog to take you to the AVD creation screen.

Setup an emulator configuration

Clicking yes on the dialog to create a new AVD takes you to the Android SDK and AVD Manager window. This is the same place you configured the SDK, but now the "Virtual Devices" panel is selected. From here, you'll be able to create a new AVD.



Test Drive

Now that you have an emulator configuration set up, run the app again. Run it the same was as before by pressing the play button in the toolbar. This will first launch the emulator and automatically install your app on the emulator and start your app.



Cool! Your first working app ...



The Android Emulator Exposed

This week's interview:
Getting to Know the Emulator

Head First: Hey there, Android Emulator. I wanted to start by thanking you for joining us tonight.

Android Emulator: Well, since I am software I do have to do what you tell me. Just kidding! Happy to be here, as always.

Head First: Fantastic! Just to clear the air here, there's been some confusion out in the development community. Are you a real Android device or, dare I say, an imposter?

Android Emulator: I'm neither, actually. I'm not a hardware device, but I'm as close to one as you're going to get with pure software.

Head First: If you're not a real device, why exactly should we use you?

Android Emulator: There are some serious benefits to me being fully software. For starters, it's easy to quickly test and debug your software without having to carry around a hardware device. Plus, since I'm fully virtual, I can run as **different** devices at the same time. If you didn't use me you'd have to carry around a *bag of phones*!

Head First: Sounds complicated. How do you keep it all straight?

Android Emulator: Well that's exactly what the emulator configurations are for! They tell me everything I need to know, from hardware configuration (like screen size), and device capabilities (like wireless latency), and even the version of Android. Everything I need to know about what device I'm supposed to act like is right there!

Head First: Neat! So not only is it easier to use you than a real device for testing, but I can test on all different kinds of devices and Android versions using you instead of keeping a stack of Android devices around!

Android Emulator: Precisely my friend. Precisely.

Head First: That all sounds great, but if there's one thing I've learned it's that nothing is ever that easy. What's the catch?

Android Emulator: The catch is that since I'm not a ***real*** device, there are some subtle differences in how I work than a real hardware device.

Head First: For example?

Android Emulator: Well, GPS is a good example. When I'm running, I sort of spoof a location based on your computers location, but I'm not really using GPS, so I can't be your only test. Photos are another good example. I don't have my own camera, so I have to fake it a little.

Head First: Sounds like mostly hardware specific differences.

Android Emulator: Pretty much. I am emulating Android hardware devices after all.

Head First: I think I've got it. You're really useful for basic testing, with a number of different configurations. But if I need to test something hardware specific, nothing beats real world hardware.

Android Emulator: Bingo!

Head First: Great. Thanks for joining us! Now, don't you have some apps to run?

Android Emulator: Sheesh! Always making me work! Anyway, always a pleasure. I'm off to help more developers test their apps!

Let's get some feedback!

You've just got your first (although pretty boring) app up and running. Before going on, let's get some quick feedback.

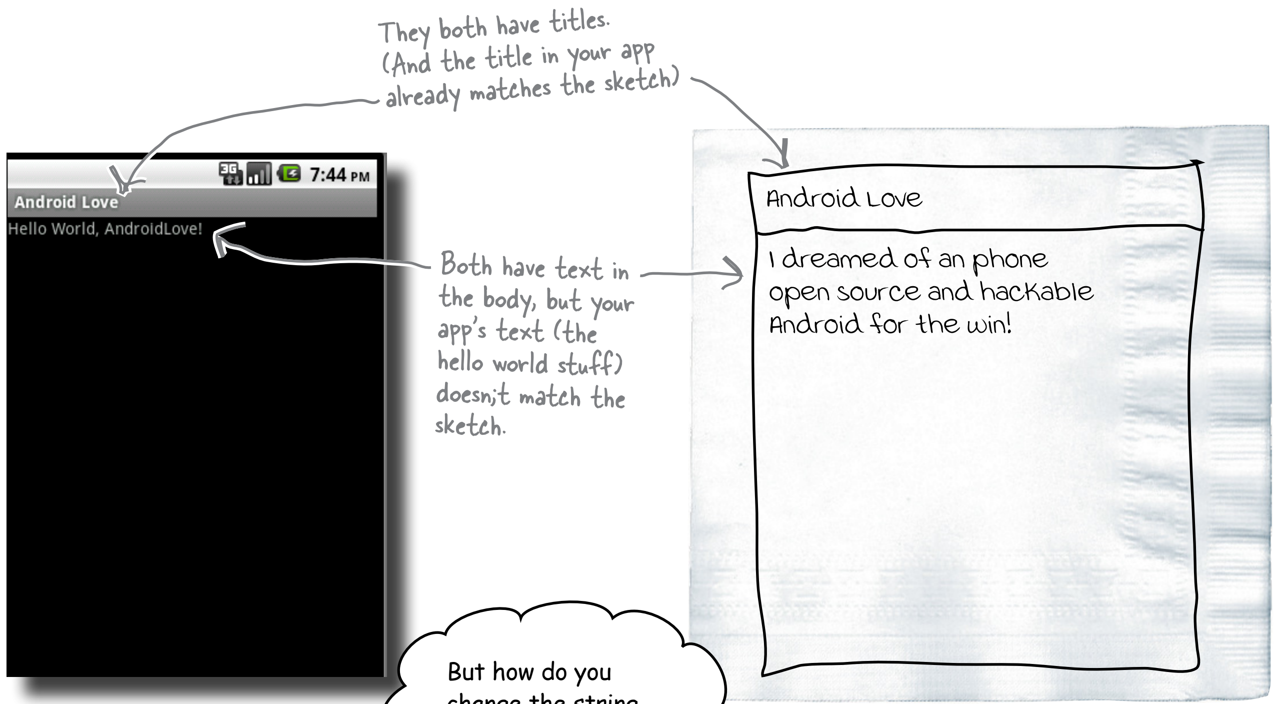


It's OK. You're not that far off...

OK, it's true. Your app isn't displaying a haiku. But take a step back and compare the app you have with the app that was sketched out. You'll see they are pretty close.

Check for differences

The app you have and the sketch for the app you want are pretty similar. The only difference is that the main text display is displaying a boring hello world message instead of the haiku. Now you just need to replace the boring string with the haiku and you'll be done with the app.





Start by looking at the layout

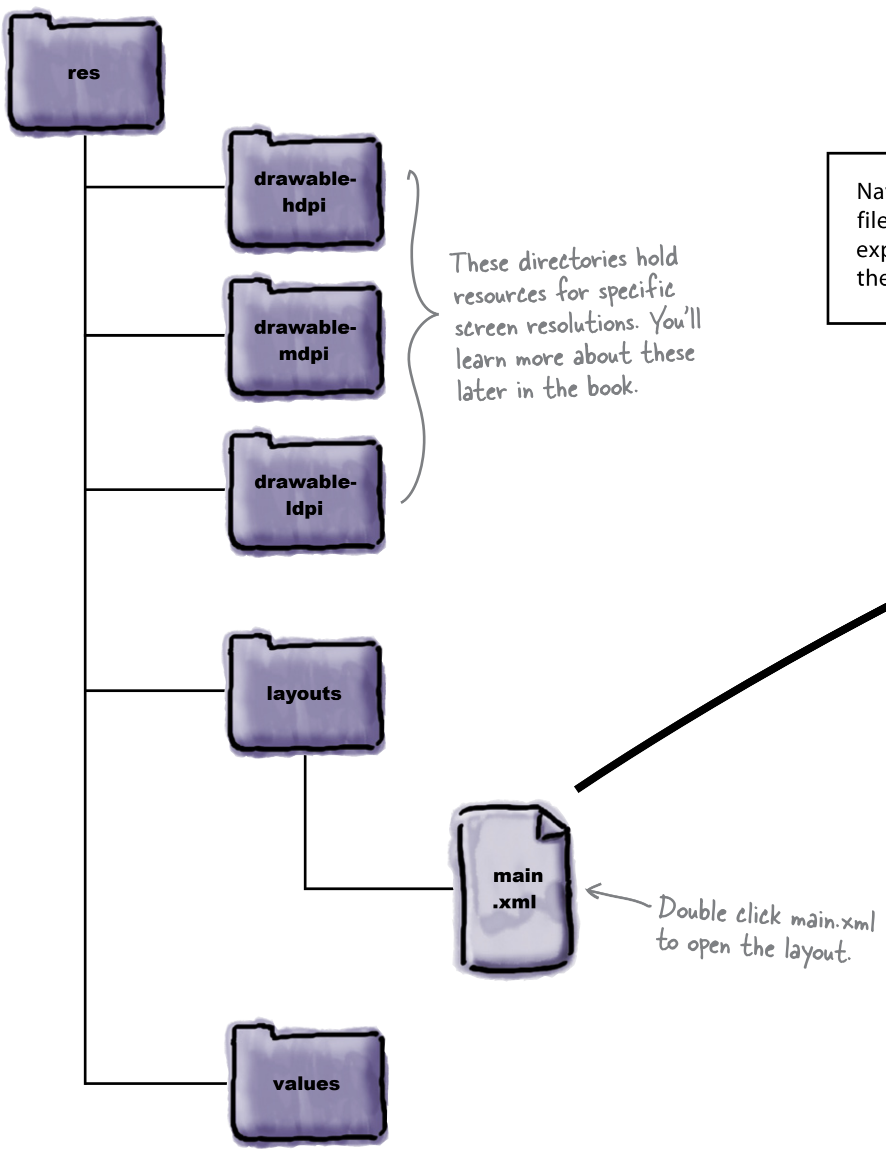
There is an XML layout that was generated by the wizard. This is what control the visual display of your app. Let's take a look at the layout and locate where the string is being set.

Locate the layout

Android layouts are defined in XML There was one layout created for you by the wizard called `main.xml`. Navigate to `/res/layout/main.xml` in the exlipse package explorer and double click on it.

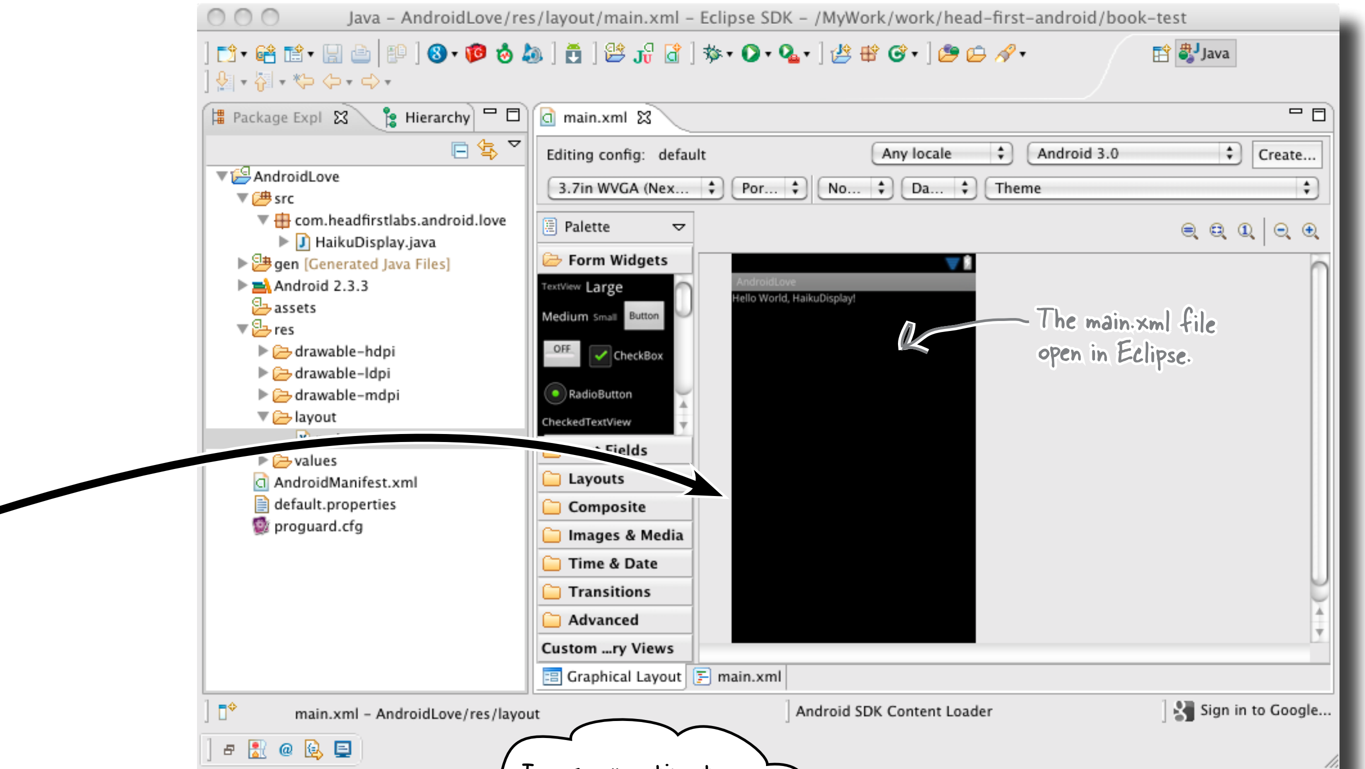
Do this!

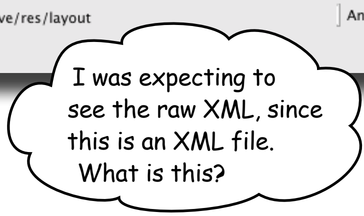
Navigate to the `main.xml` file in the Eclipse package explorer. Double click on the file to open it.



View the layout

When you double click `main.xml` and open it, you'll this new pane opened up in Eclipse.







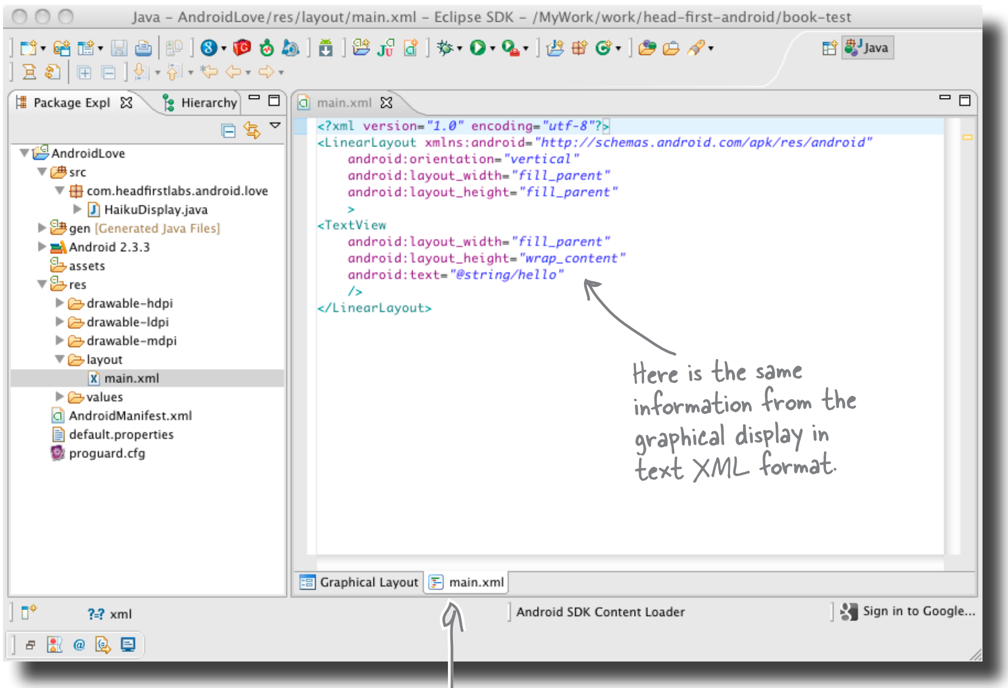
This is a graphical editor provided by the ADT

Many of the files used to build your Android apps are XML based. The ADT Eclipse plugin includes graphical editors for these files that help you edit them.

Now that you've seen the visual representation of the XML layout, you can also view the raw XML that the editor is displaying...

The layout XML

The graphical editors are just a facade over the XML underneath. So don't worry, if you want feel all super-coder, you can always jump in edit the XML source. Or you can use the graphical editors, or a mix of both!



Click the main.xml tab on the bottom to view the XML.

there are no Dumb Questions

Q: Can I edit the XML text here, or do I have to use the graphic editor.

A: The graphical editor just graphically displayed the contents of the XML text file. If you update the XML code, Android will keep the graphical editor in sync.

Q: Can I use both the graphical editor and the text editor, or do I have to choose?

A: Sure you can use both! If you make changes in the graphical editor and switch to the text view, you'll see your changes. Likewise, if you make changes in the text and switch to the graphical view, you'll see your changes there too! So` switch back and forth as much as you like!

A closer look at the layout XML

Android XML layouts consist of a number of user interface components called ***Views***, and layout managers called ***ViewGroups***. The generated `main.xml` layout has one `ViewGroup` with a single View inside it.

The main.xml layout XML code.

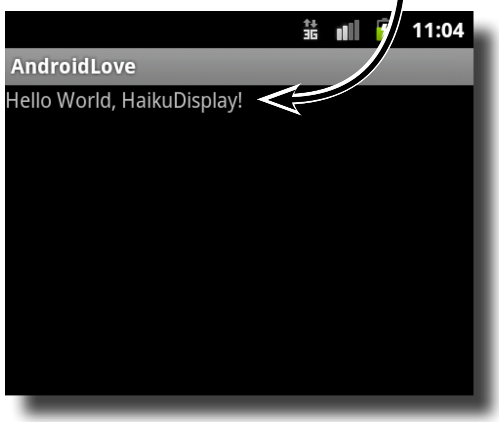
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
      xmlns:android="http://schemas.android.com/apk/res/android"
      android:orientation="vertical"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent" >
   <TextView
         android:layout_width="fill_parent"
         android:layout_height="wrap_content"
         android:text="@string/hello"
         />

</LinearLayout>
```

The View inside the layout is a TextView, a View specifically made to display text.



main.xml



The ViewGroup, in this case a LinearLayout fills the screen.

Since the TextView is displaying text, the String must be set in there somehow. Let's take a closer look...

Take a closer look at the TextView

Android Views are declared in XML layouts along with a number of attributes to configure them. Let's take a look at the generated TextView from the layout and look at it's properties.

The TextView declaration from main.xml

```
<TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello"
        />
```

These XML properties define the width and height of the view.

This attribute sets the text on the view.



Hold on, not so fast! The property seems to be setting the TextView's text to "@ string/hello" but the app says "Hello World, HaikuDisplay!". What gives?

Android loves resource properties

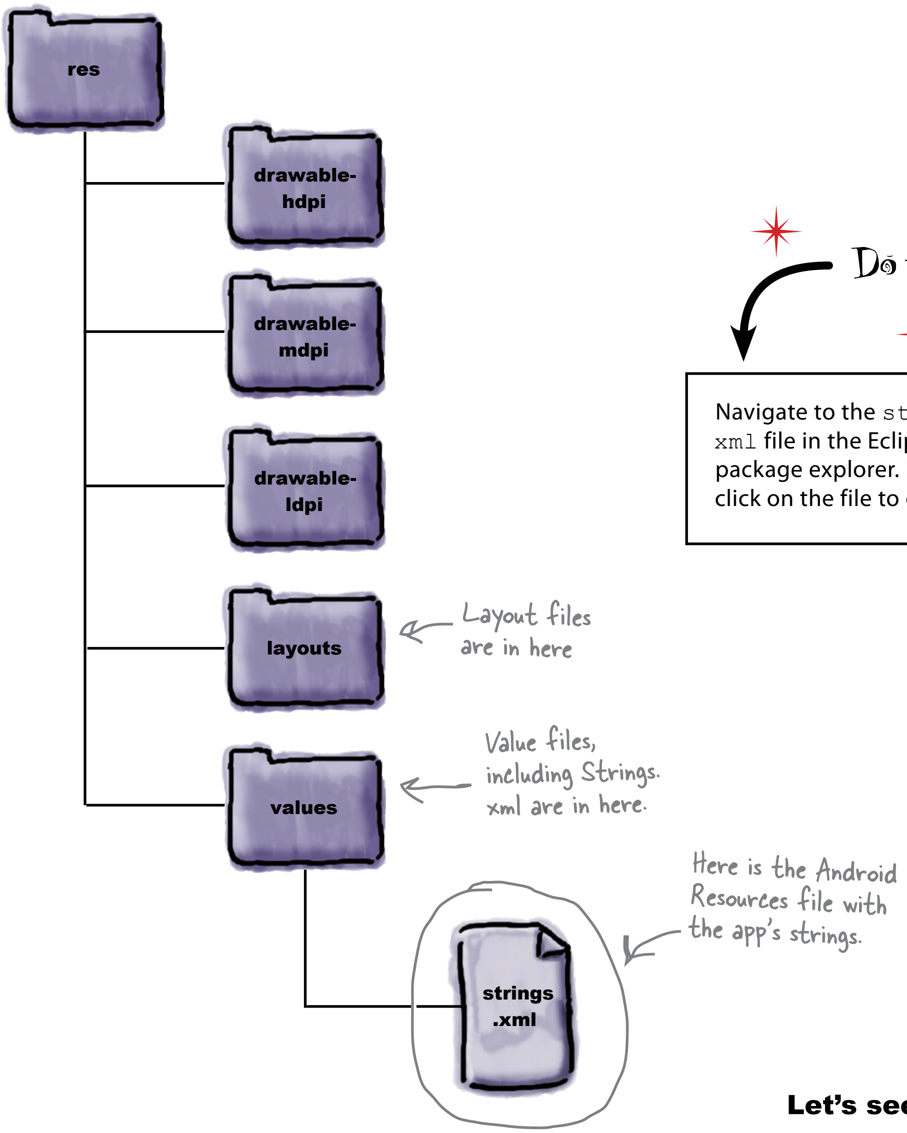
It's a good practice to move details of your user interface to property files. Developers have long since done this with text strings in their apps to spell check easier or prepare for internationalization. Similar needs hold true for colors, font sizes, image names and more!

The "@string/hello" isn't the string itself, but rather a pointer into a String property file.

Now look at the property files and locate the String definition.

Android value files

Right below layouts in the res folder is a folder called values. This folder contains the Android resource value files for your app. Open the folder and you'll see a single file named `strings.xml`. Double click `strings.xml` to open it.



Do this!

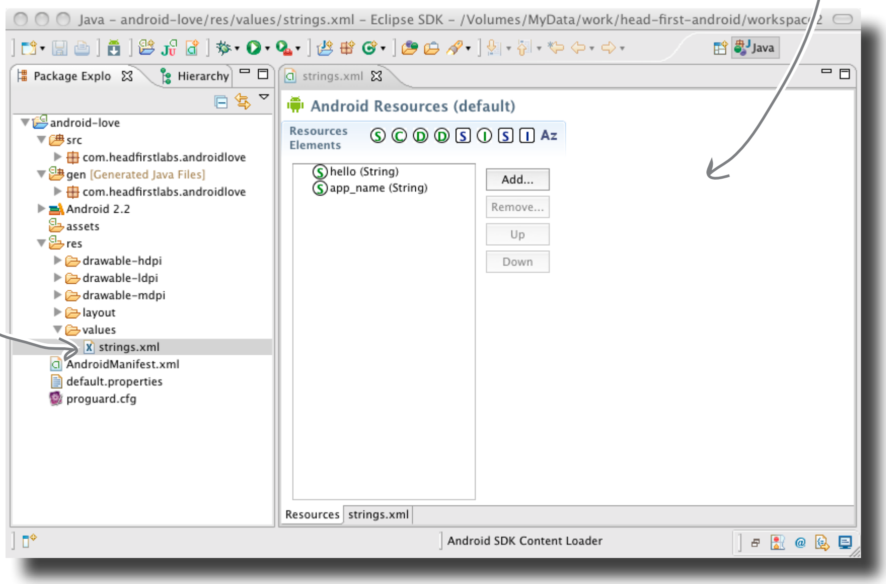
Navigate to the `strings.xml` file in the Eclipse package explorer. Double click on the file to open it.

Let's see what's inside...

Open the strings.xml file

Opening the `strings.xml` file will display another Android graphical editor in the main Eclipse pane. This editor is similar to the graphical layout editor, except that it display Android resources.

If you haven't already, navigate to the res/values/strings.xml file in the Eclipse package explorer. Double click on the file to open.



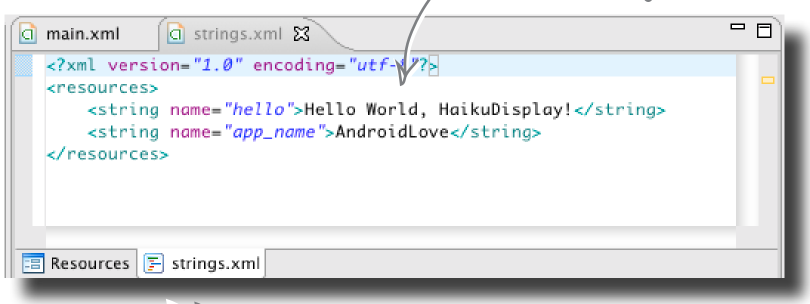
The strings.xml file opened in Eclipse.

Geek Bits

Just another graphical editor

This is just another Android graphical XML editor. Click on the tab on the bottom right to view the raw XML if you want. This works with all XML file graphical editors.

The raw XML showing name/value strings resources.



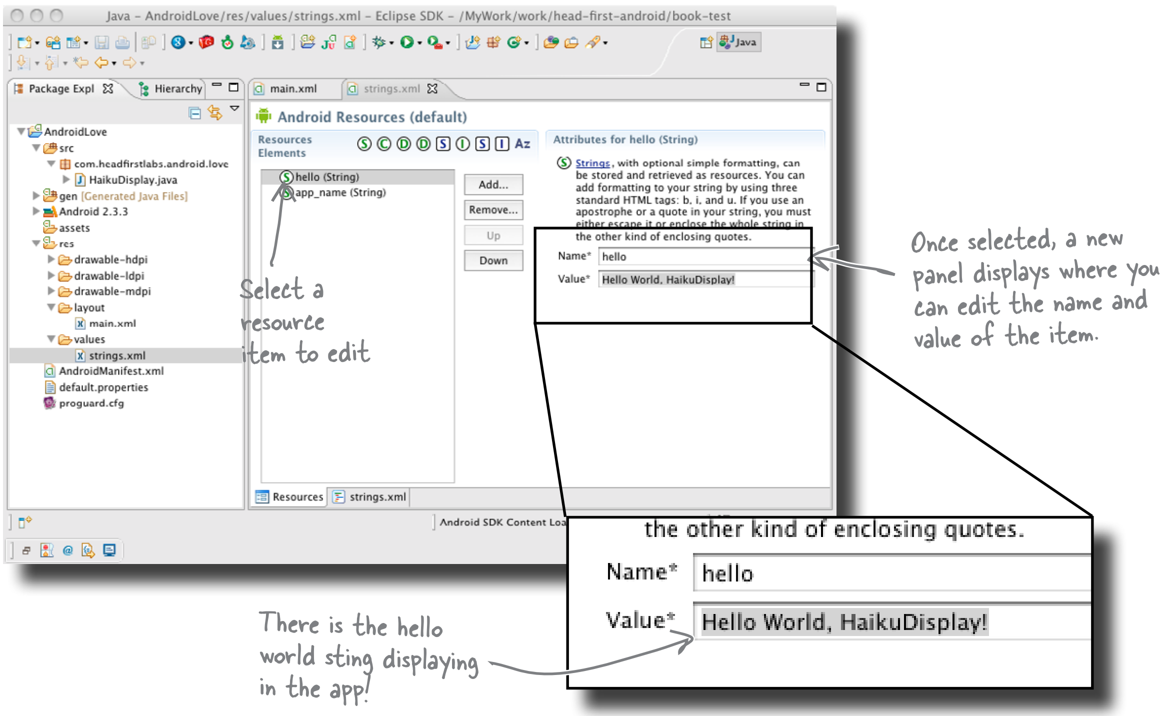
Click the strings.xml tab to view the XML.

Look at the values

You can edit any of the values by select an item from the list on the left of the pane. Once you select an item, a second panel will display showing the name and the value for that item.

Do this!

Select the first element labeled “hello” from the list.

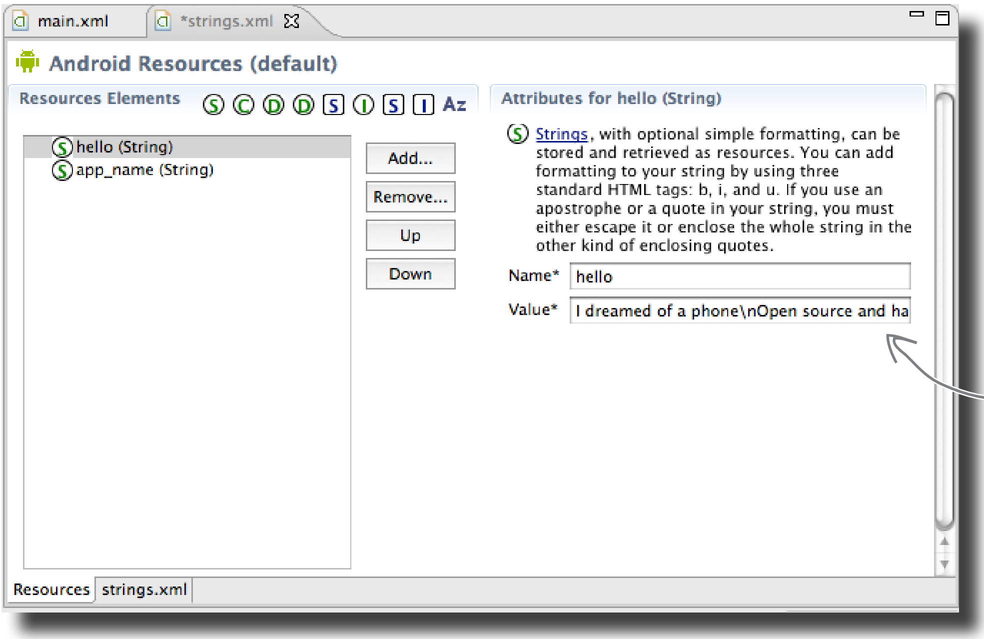


BRAIN POWER

Now that you see where the string is located, where can you edit it? Can you edit the string in the graphical editor? In the raw XML?

Edit the string

With a resource selected from the Resource Elements list, the name and value are editable on the right panel (In this case labeled "Attributes for hello (String)". Edit the "hello" Resource Element's value to the haiku.



The attribute name and value have editable text fields. Changing them here will update the value in your app.

Do this!

Edit the Value of the hello Resource Attribute with the following text "`I dreamed of a phone\nOpen source and hackable\nAndroid for the win!`". (The `\n`'s make new lines so the haiku will display on three lines.)

Watch it!

Remember to save your files.

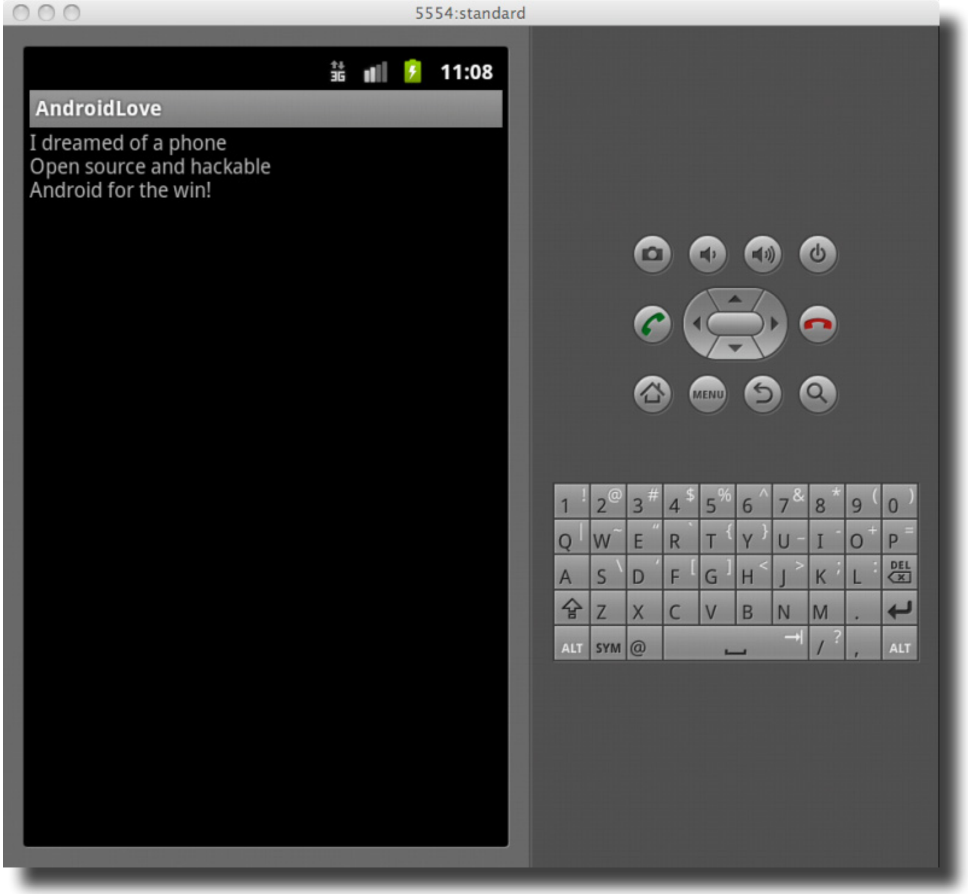
When you edit an XML file in an Android graphical editor, it generates the underlying XML. But that underlying XML is just like any other kind of text file to Eclipse and has to be saved after editing. After you make changes in a graphical editor, make sure to save before you run.



Test Drive

With the “hello” Resource Element updated with the poem, run the app again and make sure it shows your changes.

There’s the haiku displaying in the app!



Great job! The haiku is displaying in your app.

You're off to a great start!

You built your first app using the tools Google provides to help you get started quickly. Your development environment is up and running with Eclipse, the ADT plugin, and SDK configured to use an up-to-date Android version. And you modified the basic generated app to make it your own.

Stay tuned for a new feature that **Pajama Death** want toadd to the app...





Your Android Toolbox

Now that you built your first Android app, you're starting to build your toolbox of Android skills!

Installation Check List

- Install Eclipse (if you don't have it installed already).
- Install the Android SDK.
- Install the ADT Eclipse Plugin.
- Install the SDK packages.
- Configure the ADT.
- Build your awesome Android app!

Project Contents

- Screen layouts and resources (defined in XML)
- App behavior (defined in Java source code)
- Binary assets (like images and fonts) included directly in the project
- Configuration files (mostly XML)

BULLET POINTS

- Get your Eclipse-based Android development environment up and running!.
- It's a good idea to add the SDK directory to your path (while you're in a configuration mindset) so you can easily run Android tools later from the command line.
- Setup an emulator configuration for you target Android version. And don't limit yourself: feel free to setup a bunch of them!
- Create new Android projects using the Eclipse "New Android Project" Wizard. From there, modify the generated app to make it your own.
- Layouts are defined in XML and you can find them in `/res/layouts`.
- Values (like strings) are defined in Android Resource XML files. They can e found in `/res/values`.
- When you open an Android XML file in Eclipse, you'll see a graphical editor to help you modify these files. If you want to view or edit the raw XML text, click on the right tab on the bottom of the editor.
- You can go back and forth editing XML files in the graphical editor or text. Just remember to save your files when you use the graphic editor!

2 give your app an action

Adding behavior



Apps are interactive! When it comes to apps, it's what your ***users can do*** with your apps that make them love 'em. As you saw in Chapter 1, Android really *separates* out the **visual definition** of your apps (remember all that XML layout and String resource work you just did!) **from the behavior** that's defined in *Java code*. In this chapter, you're going to ***add some behavior*** to the AndroidLove haiku app. And in the process you'll learn how the XML resources and Java work seamlessly together to give you a great way to build your Android apps!

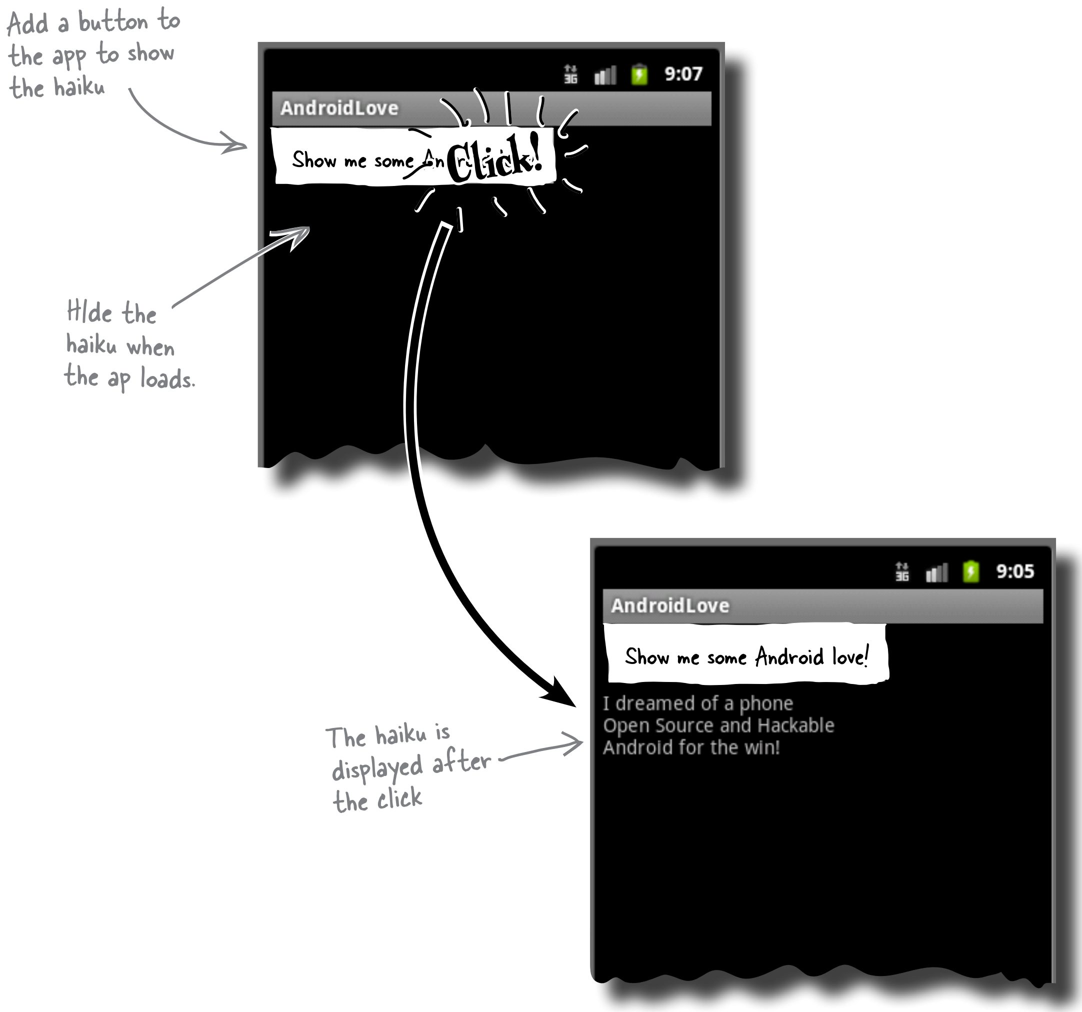
Make your app interactive



Let's see what Pajama Death have in mind...

The Pajama Death App update with an action button

Pajama Death sketched out what they were thinking so you could build it. They added a button on top of the haiku, and hide the haiku on launch. Then when you push the button the haiku shows up!



If you're thinking this looks great, but you have no idea where to start... turn the page!

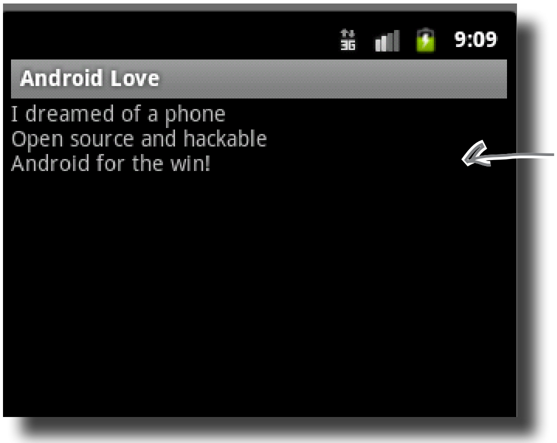
Here's how you're going to do it

You've got some work to do. *So let's break it down into a few steps.* First off, you'll be starting with the AndroidLove app project form Chapter 1, and making a few modifications to it.

Open the Android Love project now if you don't still have it open from Chapter 1.

Do this!

Open the AndroidLove project from Chapter 1 if you don't already have it open.

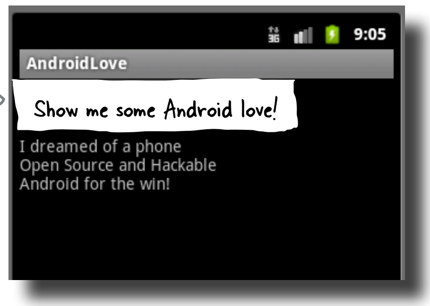


The AndroidLove app as you left it at the end of the last chapter.

1. Add the button

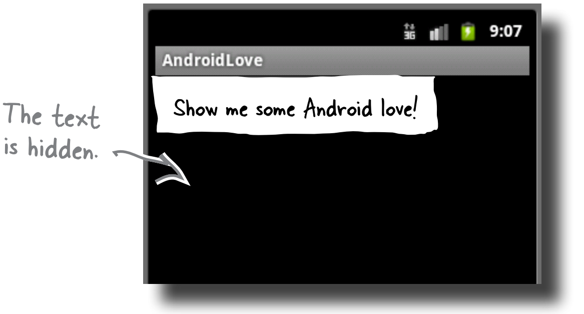
You're going to add a new button to your app's screen. Eventually, this button will show the haiku, but not in this first step. This is the first time you'll be adding a brand new component to a screen and you'll learn what components are available and how to add them to your app screens.

The new button.



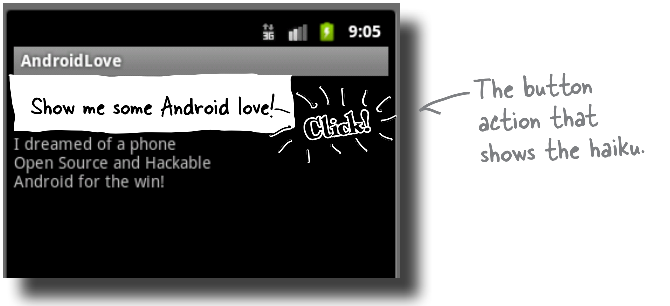
2 Hide the haiku text

After adding the button, you're going to hide the haiku text. The button still won't do anything and you won't see the haiku text at all, but hey, you're making progress! Here you're going to learn about the different attributes you can set on your widgets from XML.



3. Make the button show the haiku

Next, you're going to wire up the button to show the haiku. This is going to be your first taste of Java coding as you connect the Java behavior to the XML screens. This is where the magic happens!



BRAIN POWER

You've got your project open and you're ready to start working on this new action. The first step is adding the button. Which file do you need to open to add the button?

Add the button

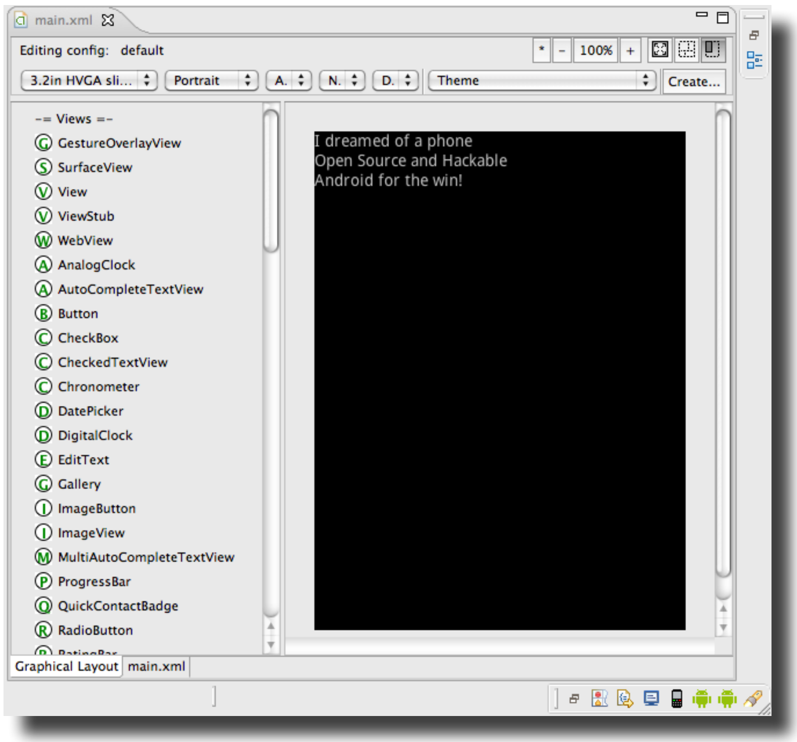
You worked with the main.xml layout file in Chapter 1 that defines the entire layout for your app's screen. This is where you're going to add the new button to your app. Open main.xmlby by double clicking on it. You can find it under /res/layout/main.xml.

In Chapter 1, you edited the XML layout in the raw XML source. Now you're going to add a component using the graphical editor. Click on the '**Graphical Layout**' tab to view the layout in the graphical editor if it isn't already showing. Notice all of the Views in the list on the left side of the screen.

Do this!

Open main.xml now. You can find it under /res/layout/main.xml.

These are all of the different Views available to you in Android.



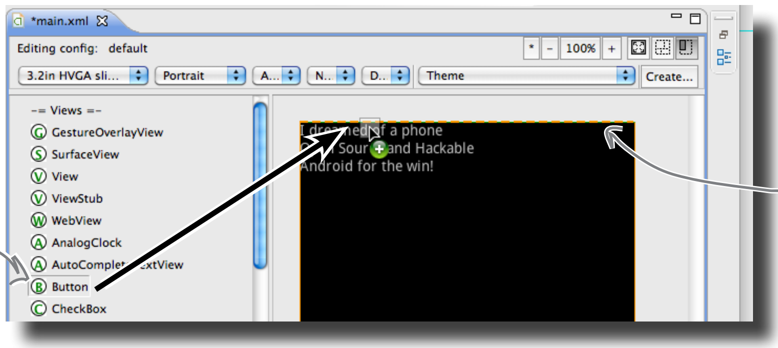
You can add views to your screen by dragging them from the list onto your screen.

Adding a View Up Close

Let's take a closer look at adding the button using the **Graphical Layout** editor.

1. Click on the button on the left panel and drag it to the top of the graphical layout. You'll notice an dotted line display where the button is going render. Make sure it goes at the top.

Click on the Button and drag it to the top of the layout.



Drag the button all the way to the top and you'll see an orange dotted line where the button will be added.

After you add the button it'll look like this.

Here's the button you just added.



2. Now click back to the `main.xml` showing the XML. You'll the first `View` defined in the file is the `Button` you just added!



The added XML declaration to create the Button.

Fix the button text

It's great that the button is on the screen now, but not so great that the button text is showing up as "**@+id/Button01**". Let's see about changing that.

Here's the button with the weirdo button text showing up as "@+id/Button01"



Why is the button text showing up like this?

To get to the bottom of this, compare the `View` XML declarations of the `TextView` displaying the hauki and the `Button` you just added. Focus on the text properties of each `View`.

The haiku `TextView` `android:text` property is referring to the haiku string property in `strings.xml`.

The haiku TextView XML declaration from main.xml.

```
<TextView android:text="@string/haiku"
      android:id="@+id/haikuTextView"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content" />
```

The android:text attribute is set to "@string/haiku" which references the haiku String resource in strings.xml



main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
      <string name="haiku">I dreamed of a phone \nOpen Source
           and Hackable \nAndroid for the win!</string>
      <string name="app_name">AndroidLove</string>
</resources>
```



strings.xml

Now look at the Button definition

The `Button` definition's `android:text` attribute value doesn't have the "`@string/`" prefix. It just has "`@+id/Button01`" as it's value.

The new Button XML declaration.

The android:text attribute is referring to Button01.

```
<Button android:text="@+id/Button01"
        android:id="@+id/Button01"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
```



main.xml

Wait a second! There is no Button01 string property in strings.xml. What gives?



The answer lies in the prefix...

The value for the `android:text` property in the `TextView` is referring to a String resource in `strings.xml`. ***But there is no string resource for the Button!***

There are string properties for "haiku" and "app_name" but nothing for "Button01"

The @string prefix

Take another look at the haiku `TextView` text attribute and you'll see it has a special prefix "`@string/`". That special prefix tells the view rendering code to look into the `strings.xml` file for a string property. And even though the `Button` has a prefix before `Button01`, it's not the special "`@string/`" prefix so it **doesn't work**.

Using the @string prefix

```
<TextView android:text="@string/haiku"
```

The TextView's text has the "@string/" prefix.

NOT using the @string prefix

```
<Button android:text="@+id/Button01"
```

The Button doesn't have the special "@string/" prefix.

there are no Dumb Questions

Q: If the Button is missing the `@string` prefix, how is it displaying any text at all?

A: If the Android view rendering code doesn't detect the `@string` prefix to look up a key in the `strings.xml` file, it renders the value in the `android:text` directly.

Q: Is that why the button says "`@+id/Button01`" because it's rendering directly from the `android:text` property?

A: Exactly.

Q: Hey cool! So why are we messing with strings.xml file at all? Couldn't I just put all of my strings directly in the layouts and call it a day?

A: Technically, yes. But it's not the best idea. The string resource element was designed to remove string constants from your layouts. It's a good idea to keep them separate, and Android is setup to handle this out of the box.

Add a string resource for the button

The fix for this is going to include two changes. You'll need to add a new string property in `strings.xml`, and then you'll need to update the `Button` definition in `main.xml`.

Let's start by adding the new string resource. Open `strings.xml` and click on the `strings.xml`. This is where you're to add the new `String` property and you'll do it directly in XML!

Here is the format.

Start the element with String. This is so android knows it's a String resource.

Give it a name, that's what you'll use to reference this string in your layout.

```
<string name="haiku">I dreamed of a phone \nOpen Source
       and Hackable \nAndroid for the win!</string>
```

The value is the actual string you want to display.



Below is the the contents of the strings.xml file. Add a new String property called "love_button_text" and give it a value of "Show me some Android love!"

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
       <string name="haiku">I dreamed of a phone\nOpen Source
              and Hackable\nAndroid for the win!</string>
       <string name="app_name">AndroidLove</string>

       ..............................................................

</resources>
```

Add the new property here.

Exercise Solution

Below is the the contents of the strings.xml file. You should have added a new String property called “love_button_text” and given it a value of “Show me some Android love!”

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
       <string name="haiku">I dreamed of a phone\nOpen Source
              and Hackable\nAndroid for the win!</string>
       <string name="app_name">AndroidLove</string>
       <string name="love_button_text">Show me some Android love!</string>
</resources>
```

The element is a String element.

The element has a name attribute of “love_button_text”.

And the value is set to “Show me some Android love!”

Now you just need to use it!

You just added the String resource for `love_button_text`. Now it’s time to plug it into the `Button` declaration in `main.xml` to set the text.

Sharpen your pencil

Below is the `main.xml` layout. Now that you have the love_button_text property, use it in the `Button` definition to set the text form the `strings.xml` resources.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
      xmlns:android="http://schemas.android.com/apk/res/android"
      android:orientation="vertical"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent">

<Button android:id="@+id/Button01"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="..............................................."
      />

<TextView android:text="@string/haiku"
      android:id="@+id/haikuTextView"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content" />

</LinearLayout>
```

Use the "@string/" prefix plus the String resource name here to have the Button reference the String resource you just added.

Sharpen your pencil Solution

Below is the `main.xml` layout. Now that you have the love_button_text property, you should have used it in the `Button` definition to set the text form the `strings.xml` resources.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
      xmlns:android="http://schemas.android.com/apk/res/android"
      android:orientation="vertical"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent">

<Button
      android:id="@+id/Button01"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/love_button_text"
      />
```

Here's the prefix telling the view rendering to use a String resource

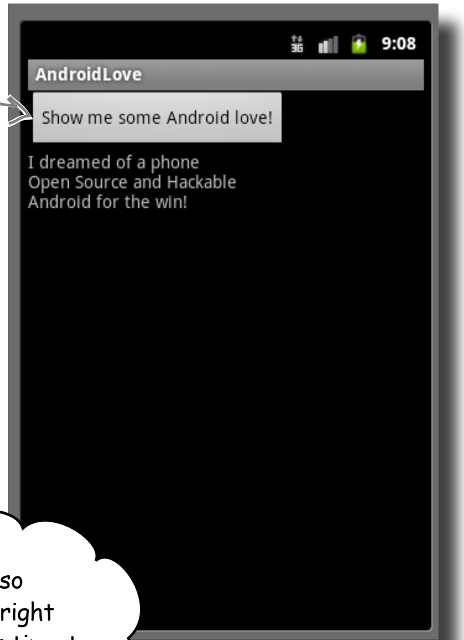
And here's the name of the String resource to use.

```
<TextView android:text="@string/haiku"
      android:id="@+id/haikuTextView"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content" />
</LinearLayout>
```

Test Drive

Whew! You added the `Button`, which had some weird text. And to fix that, you added a new String resource, and used that new String resource from the `Button's` `android:text` attribute. Let's see if it all worked! Run the app again...

The button is displayed with the correct text from the String resource.



And it works! The button looks good!



Hide the haiku text

Now that the `Button` is added and looking good, it's time to move on to the next step: **hiding the haiku text**.

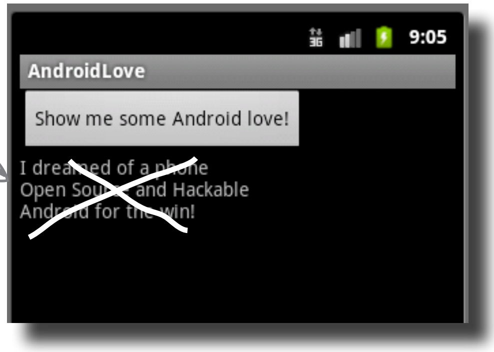
How are you going to do this?

Well, two strategies are probably coming into your head right now. You could remove the `TextView` and it back once the button is pushed or you could set the text to be invisible and make it visible once a user presses the button.

Let's go with the invisible text option!

OK, but that's not a huge help, right? You need to know how to hide text. This is something new that you haven't done yet and you need to know where to find out about new things in Android. Luckily, Android comes with great online documentation for just this reason! You can view is at `developer.android.com/reference`.

You need to hide this text.





Do this!

Go to the online Android documentation now at `developer.android.com/reference`.

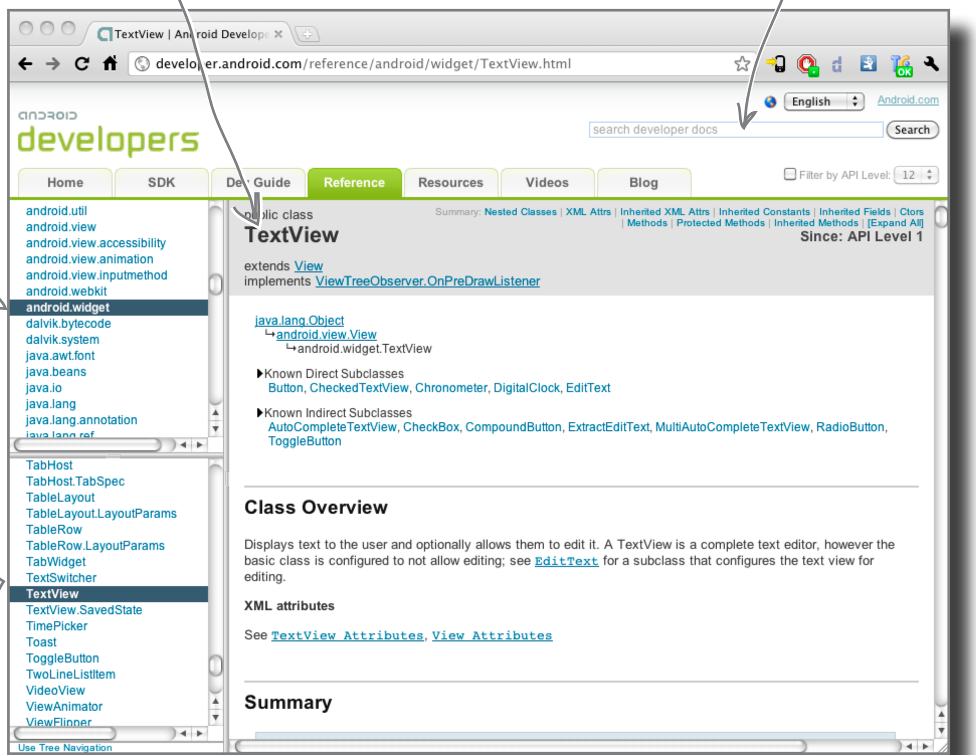
Documentation Navigation Up Close

Let's take a quick look around the Android online documentation to get acquainted. You can navigate to what you're looking for by either selecting the package and class name, or searching for a class name in the search box on the top right. Now since you're looking to update an attribute on the `TextView`, you need to look at the `TextView` documentation.

When you click on a class or a package, the main panel will show the details for what you've selected.

If you know the class you're looking for, but now the package, you can type it in here to search the documentation.

This area lists all of the packes in the documentation. Click on one to view the package documentation. In this case, the android.widget package is selected.

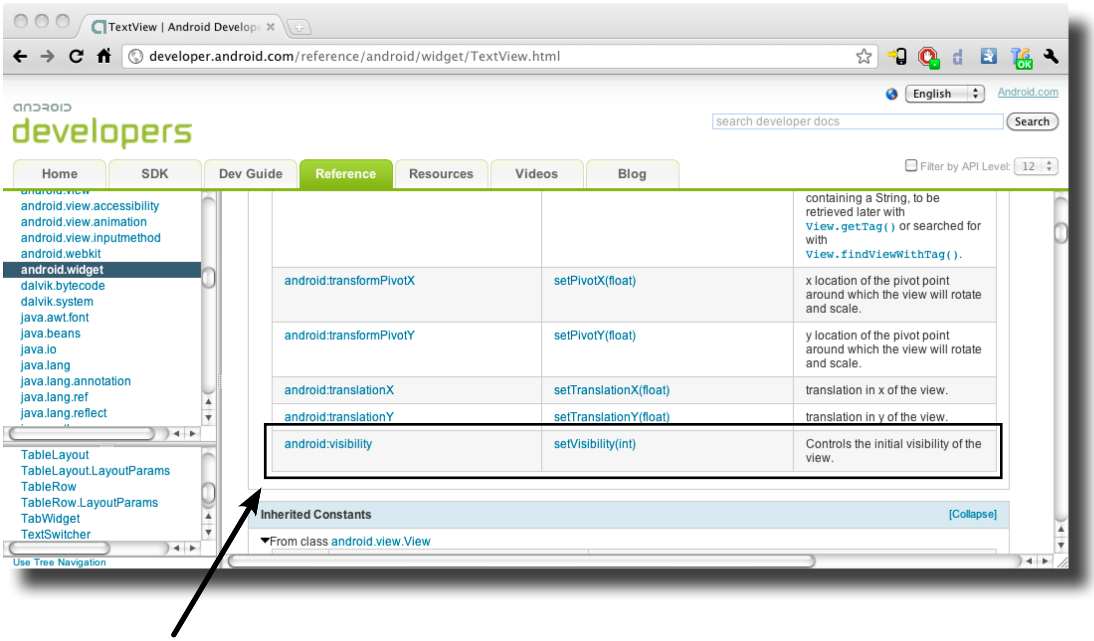


Once a package is selected, this section will show all of the classes in that package. In this case, the TextView is selected.

Browse the XML attributes

As you browse the documentation for `TextView`, you'll notice it has a number of Java methods, but it also has XML attributes listed. That's because internally, `TextView` is a complete Java class. Since you're working with the `main.xml` layout definition in XML, focus on the XML attributes.

Does any look interesting? You're looking for something that can hide the text...



This looks perfect!

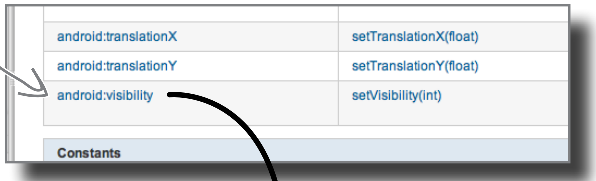
It says it can control the "visibility of a view." That's exactly what you want! Using this you can make the entire `TextView` invisible when the app starts up.

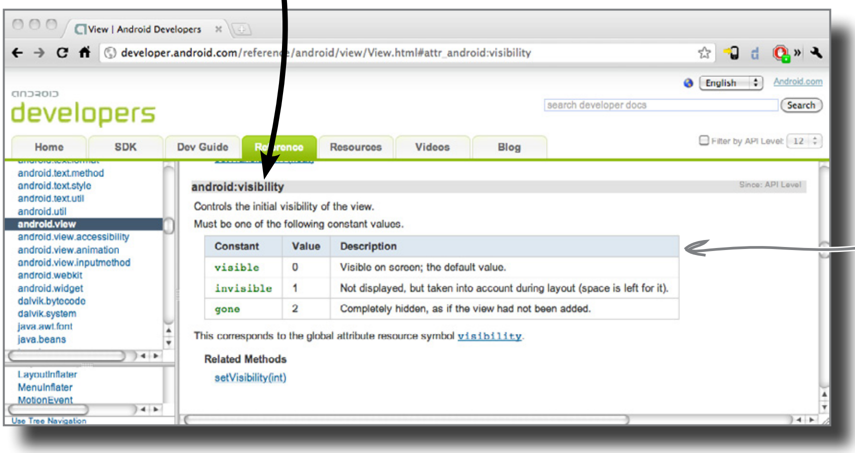
So how does it work?

View XML attribute details

If you click on any attribute, you'll be taken to a section that details the usage of that attribute. Click on `android:visibility`, you'll be taken to the detail section on it's usage.

Click here to view the usage details for android:visibility.





Detailed usage for android:visibility.

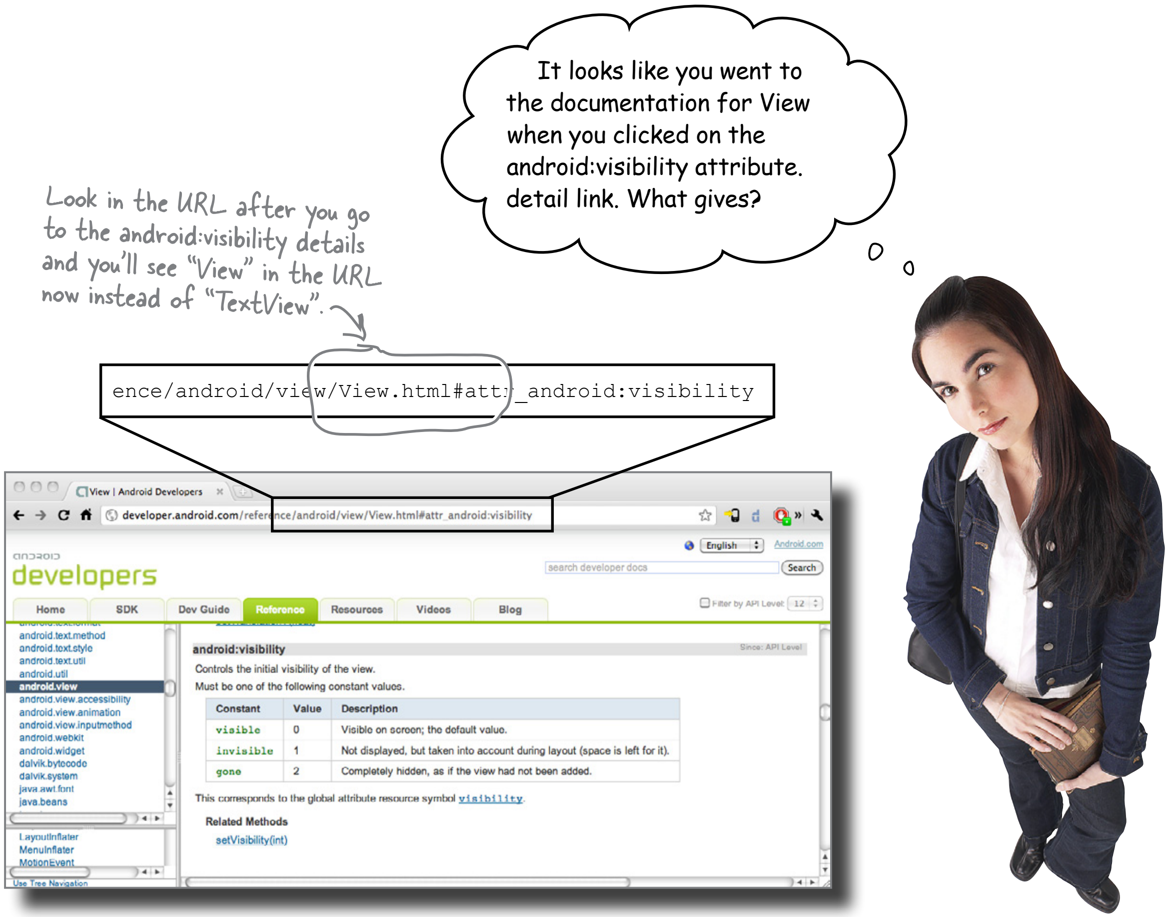
This tells us the usage is like this:

Attribute values are always in quotes.

`android:visibility` = **`"invisible"`**

This is the name of the XML attribute, which matches the name in the docs.

Use invisible since you want to hide the view.



View is a base class that other widgets inherit from

The `View.java` class is a base class with several cross widget methods, attributes, and constants. And if you look at the headers for both `Button` and `TextView`, you'll see that they both inherit from `View`. The Android docs include superclass methods descriptions along with the locally implemented methods (but if you look close you will see that the `android:visibility` attribute was located in a section called ***Inherited XML Attributes***).

Sharpen your pencil

Below is the `main.xml` layout code. Update this code with the `android:visibility` set to `invisible`. This will hide the `TextView` and with it the haiku text.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
       android:orientation="vertical"
       android:layout_width="fill_parent"
       android:layout_height="fill_parent">

<Button android:text="@string/love_button_text"
       android:id="@+id/Button01"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content" />

<TextView android:text="@string/haiku"
       android:id="@+id/haikuTextView"
       android:layout_width="fill_parent"
       android:layout_height="wrap_content"

       ..........................................................................

       />

</LinearLayout>
```

Add the android:visibility attribute here.

Sharpen your pencil Solution

Below is the `main.xml` layout code. You should have updated this code with the `android:visibility` set to `invisible`. This should hide the `TextView` and with it the haiku text.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent">

<Button android:text="@string/love_button_text"
        android:id="@+id/Button01"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

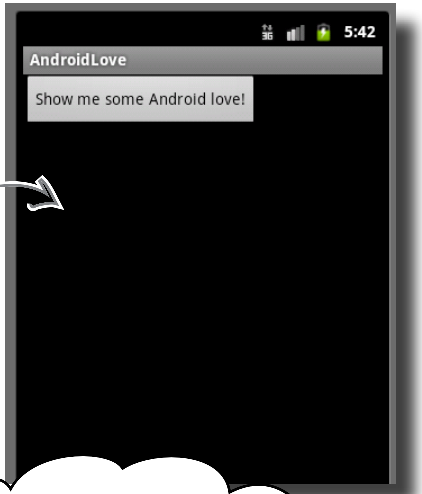
<TextView android:text="@string/haiku"
        android:id="@+id/haikuTextView"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
          android:visibility="invisible"
        />

</LinearLayout>
```

Here's the android:visibility attribute set to invisible. This should hide the whole haiku TextView!

Test Drive

You've hidden the `TextView` with the haiku on it with the `android:visibility` attribute. Now run the app and make sure it worked!



Setting the android:visivibility attribute to 'invisible" hid the text.

The text is gone. Great job!



Let's get that button working!

Make the button show the haiku

It's time to start making that `Button` work! There is an attribute on the `Button` View for just this purpose called **`android:onClick`**. The value for the attribute is the name of the action you want to use.

Let's use it now!

Add the `android:onClick` property to the `Button` definition in `main.xml`. Give is a value of `onLoveButtonClicked` to be descriptive of what the `Button` is supposed to do.

The Button definition from main.xml

```
<Button android:text="@+id/Button01"
        android:id="@+id/Button01"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="onLoveButtonClicked"
        />
```



main.xml

The onClick attribute added to the Button. Pointing to the onLoveButtonClicked method.

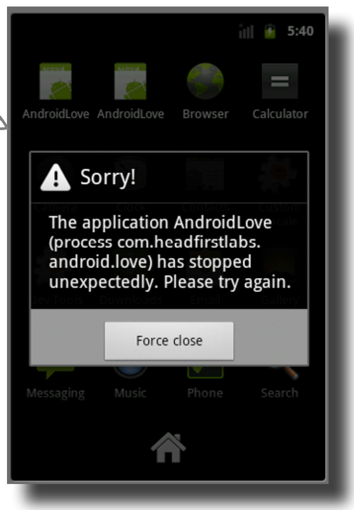
Wait a second! What is onLoveButtonClicked? Is it more XML code that you're going to define in main.xml, or somewhere else?



Actually, it's a Java method. It's just not written yet...

So far, you've updated the screen layout, added a new `View` to the screen, modified and added `String` resources. All of these changes control the way the app starts. But for the button action, you'll be making a change that a user can initiate while the app is running- adding behavior to the app. And Android app behavior is defined in Java.

You'll get an error like this if you run your app now and press the button. This is because onLoveButtonClicked isn't defined yet.

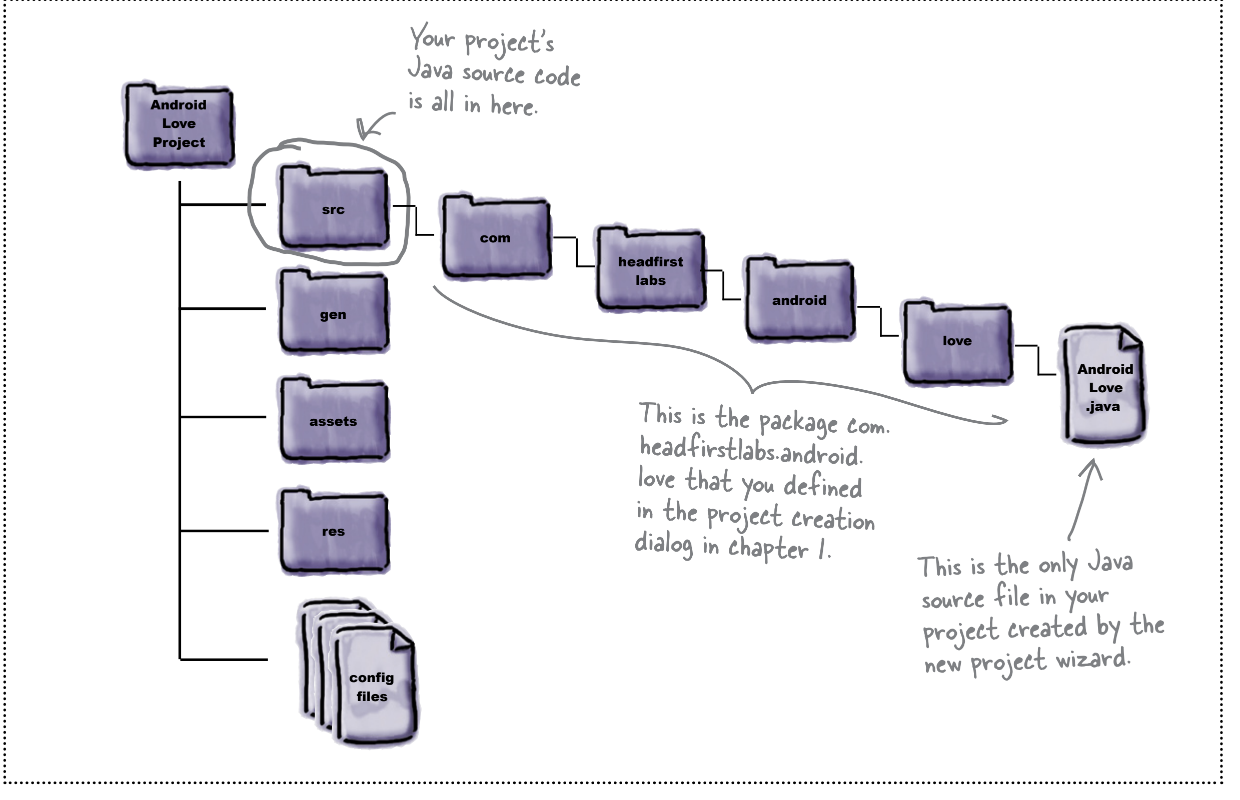


So, let's define onLoveButtonClicked now...

Defining onLoveButtonClicked

So defining `onLoveButtonClicked` in the `android:onClick` property on the `Button` is calling some kind of Java method. But where is that method supposed to go?

Let's start by taking a look at the Java source code in your project and it's contents.



Only one Java source file created by the wizard? Let's take a closer look at it...

The AndroidLove Activity

The `AndroidLove` class is a subclass of a built in Android class called `Activity`. Think of an `Activity` as the Java code that supports a screen. And in this case, `AndroidLove` is actually the `Activity` that supports your main screen you're defining in `main.xml`.

Double click on `AndroidLove.java` and Eclipse will automatically open it in a Java editor.

The source for AndroidLove.java

```
public class AndroidLove extends Activity {

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }

}
```

AndroidLove extends Activity

This code is setting the view defined in main.xml on the screen. You'll see how it works soon!



AndroidLove.java

The button is expecting to call a method in this class.

Since the `AndroidLove Activity` is setting the `main.xml` layout on the screen, the Android action code is going to look for the method defined in the `android:onClick` attribute here. The action code is going to look for a method in the following format.

The method needs to take one argument of a View. This is the view that was clicked.

```
public void onLoveButtonClicked ( View view )
```

The method name needs to match the value of the android:onClick attribute

Add the action method

Let's add the `onLoveVuttonClicked` method to `AndroidLove` now. Once this is done, we can run the app and press the button and it shouldn't break.

```
public class AndroidLove extends Activity {

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }

    public void onLoveButtonClicked(View view) {
        //doesn't do anything yet
    }

}
```

The new onLoveVuttonClicked method that's referenced from the android:onClick Button attribute.

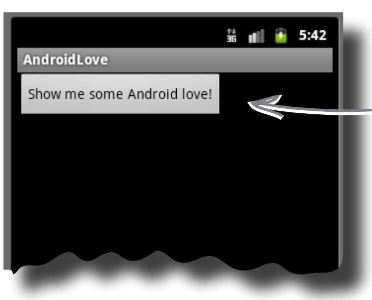


AndroidLove.java



Test Drive

Run the app now and press the button. It won't perform any actions yet. But you also won't see errors either.



You can run the app and click the button now. Nothing will happen, but the app won't force close either.

Implementing the action method

Great work so far! The `Button` has an action method configured in the `android:onClick` property (`onLoveButtonClicked`). The `onLoveButtonClicked` method has an empty implementation in the `AndroidLove Activity` which you've verified is being called since the app doesn't crash. Whew!

Now it's time to implement the onLoveButtonClicked method and make it show the text!

Implementing the action in the `onLoveButtonClicked` method really consists of two parts. First, you need to get a reference to the `TextView` and then you need to set the `visibility` property to true. *Sounds simple enough, right?*

Cool! Let's get started...

```
public class AndroidLove extends Activity {

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }

    public void onLoveButtonClicked(View view) {
        TextView haikuTextView = ?!?
    }

}
```

Make a variable to reference the haiku TextView...

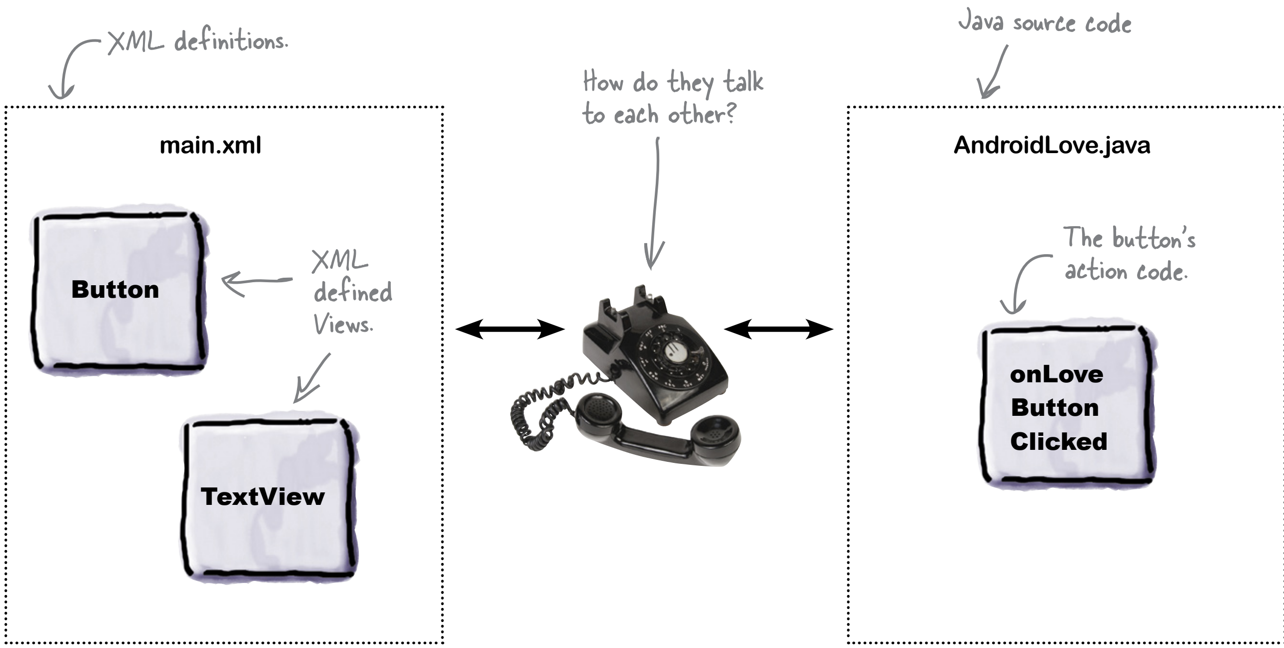


AndroidLove.java

Wait, how do you get a reference to the TextView?

From XML to Java

You've got a disconnect right now. Your screen Views (the `Button` and the `TextView` displaying the haiku are defined in XML in the `main.xml` layout. But you action code is defined in Java in the `AndroidLove Activity`. How are you supposed to get references to XML defined `Views` from your Java code?



The 'R' file

To solve this, Android generated a special called the 'R' file. This is a file of constants that allow you to get Java references to the `TextView` you defined in `main.xml` In fact, you can get references to all kinds of in app resources you define! But remember the `String` resources you defined in XML? You can get references to those too.

Do this!

Open the R file now. You can find it under `gen/com/headfirstlabs/androidlove/R.java`

The R File Way Up Close



The R file consists of a number of `public static final` constants, each one referring to an XML resouce. The constants are grouped into interfaces according to XML resource type.

Your R.java should look like this:

```
public final class R {
        public static final class attr {}
        public static final class drawable {
                public static final int icon=0x7f020000;
        }
        public static final class id {
                public static final int Button01=0x7f050000;
        }
        public static final class layout {
                public static final int main=0x7f030000;
        }
        public static final class string {
                public static final int app_name=0x7f040001;
                public static final int haiku=0x7f040000;
                public static final int love_button_text=0x7f040002;
        }
}
```

Interfaces grouping the constants.

Constants referring to XML resource.



R.java

Android provides a number of utility methods for using these constants. Take another look at the `onCreate` method from `AndroidLove.java` where the screen layout is set. `setContentView` takes an `R.java` constant which was generated from the `main.xml` layout.

```
public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
}
```



AndroidLove.java

setContentView is called with the constant at R.layout.main to set the layout defined in main.xml on the screen.

Getting view references

Setting the content view from the R file is nice and all, but what you really want to do is get a reference to the `TextView`! Well, Android provides another cool utility method called `findViewById` to do just that. The `findViewById` method is in the base class of Activity, so you can use it directly in the `AndroidLove` class since it's a subclass of `Activity`.

The `findViewById` method takes one parameter, the R constant for the `View`. But since the method is meant to be generic, it returns a `View` not one of the `View` subclasses (like `Button`, `TextView`, or any other `View`). It's easy enough though, you just need to cast the result to the `View` you're expecting.

Let's see how this works for retreiving a reference to the button on screen.

View R constants are in the 'id' interface group

Make a reference to store the returned View

Cast the returned View to the appropriate View class you're looking for.

Pass the R.id.Button01 to findViewById to get a reference to the on screen button.

```
Button button = (Button) findViewById(R.id.Button01)
```

```
public final class R {
        public static final class attr {}
        public static final class drawable {
                public static final int icon=0x7f020000;
        }
        public static final class id {
                public static final int Button01=0x7f050000;
        }
        public static final class layout {
                public static final int main=0x7f030000;
        }
```



R.java

Give the textview an id

Take another look at the id interface inside `R.java`. There is a constant for the Button but not for the `TextView`. **Weird, huh?**

The issue here is that the R file constants for the `Views` are generated based on an `android:id` attribute in `main.xml`.

```
<Button android:text="@string/love_button_text"
        android:id="@+id/Button01"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="onLovebuttonClicked" />
```

This android:id attribute controls the name of the constant created for the Button in the R file.

```
<TextView android:text="@string/haiku"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:visibility="invisible" />
```

There's no android:id attribute defined in the TextView declaration so no R file constant is created.



main.xml

Sharpen your pencil

There"s no `android:id` attribute defined in the `TextView` declaration in `main.xml`, so no R file constant get's generated. Don't worry though, you can just add one yourself! Below is the the `TextView` declaration in `main.xml`. Add an `android:id` attribute and give it a value of "haikuTextView"

```
<TextView android:text="@string/haiku"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:visibility="invisible"
        ..................................................................
/>
```

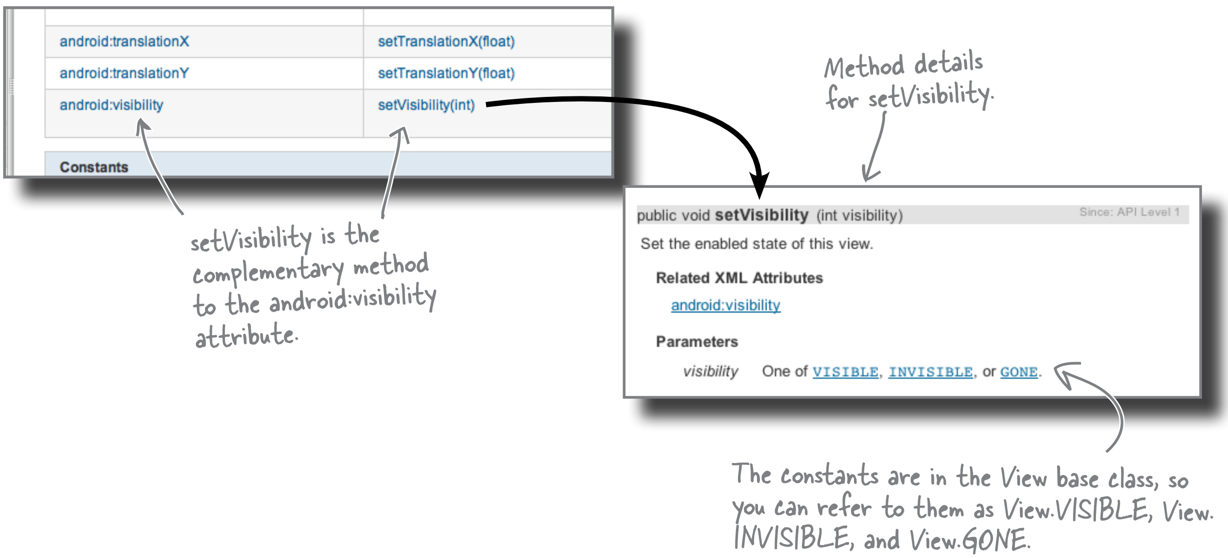
Sharpen your pencil Solution

There wasn't an `android:id` attribute defined in the `TextView` declaration in `main.xml`, so no R file constant get's generated. Below is the the `TextView` declaration in `main.xml`. You should have added an `android:id` attribute and given it a value of "haikuTextView" so an R file constant will getgenerated.

```
<TextView android:text="@string/haiku"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:visibility="invisible"
        android:id="@+id/haikuTextView"
/>
```

Complementary Java methods

Most of the properties you can set from XML can also be set from code. This is important since you need to make the haiku `TextView` visible from the `v` action in Java. Let's take another look at the `TextView` documentation for `android:visibility` and look for the complementary Java method.





The Complete Action Magnets

You've got all the pieces you need to write the `onLoveButtonClicked` method now! Below is the code for the `AndroidLove` Activity, but the method is `onLoveButtonClicked` blank. The magnets below contain all of the code fragments you need to finish the method. Use the magnets to complete the implementation.

```
public class AndroidLove extends Activity {

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }

    public void onLoveButtonClicked(View view) {
        ...........................................

        ...........................................

        ...........................................
    }
}
```



AndroidLove.java

This is a constant you can pass into setVisibility to make the View visible.

```
textView.setVisibility(
);
R.id.haikuTextView
TextView textView =
);
(TextView) findViewById(
View.VISIBLE
```



The Complete Action Magnets Solution

Below is the code for the `AndroidLove` Activity. The magnets below contain all of the code fragments you needed to finish the `onLoveButtonClicked` method. You should have used the magnets to complete the implementation.

```
public class AndroidLove extends Activity {

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }

    public void onLoveButtonClicked(View view) {
        TextView textView = (TextView) findViewById(
            R.id.haikuTextView );

        textView.setVisibility( View.VISIBLE );

    }
}
```

Get the TextView reference using the R constant.

Set the TextView visibility to true so it's displayed.

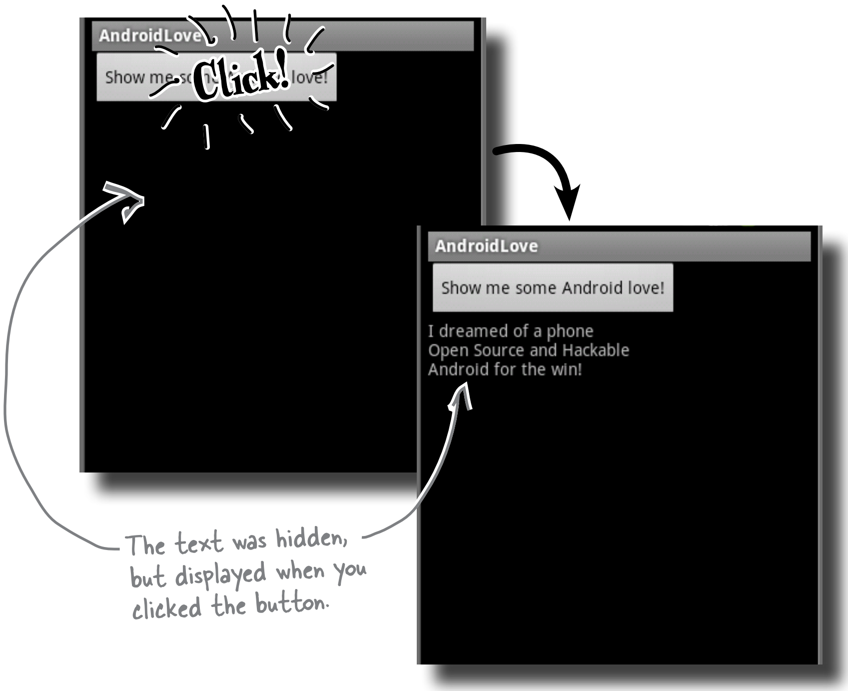


AndroidLove.java

Let's run it now!

Test Drive

Now that the `onLoveButtonClicked` method is complete, run the app and try it out.



You did it!

When you started the chapter, the AndroidLove app had no behavior, it didn't **do** anything. But now you've made it do something! And to make that happen, you added a new view, created and used a new string resource for it's text, built a button action in Java, and used the R file to help go back and forth between Java and XML.

Great work!





Your Adding Behaviour Toolbox

Now that you've completely implemented a button action, you can start adding behavior to all your apps!

Making a Button Action

- Use the Button onClick attribute to declare the name of the action method
- Open the Activity that displays the layout with the Button
- Add a method with a name matching the onClick attribute value
- Make sure the method takes in a single View as a parameter

- Use the graphical layout editor to make adding new Views easy.
- Add new String resources when you need them (and add them to Strings.xml).
- Use the "@string/" prefix in your XML layout to refer to String resources.
- Explore the online documentation for all of the attributes you can set in your XML layouts.
- If you know what you're looking for but don't know where to find it, use the documentation search
- Get references to Views on screen by calling `findViewById` and passing in that `View`'s ID constant from the R file.
- Make sure your Views in your XML layout have android:id attributes set if you need to get references using `findViewById`.
- to use Use the `android:onClick` property on `Button` to add an action method. That action method will be called on the Activity that launched the screen, so make so to add the method.
- Remember all of the Java source is in the `/src` folder.

3 working with feeds

Pictures from Space!



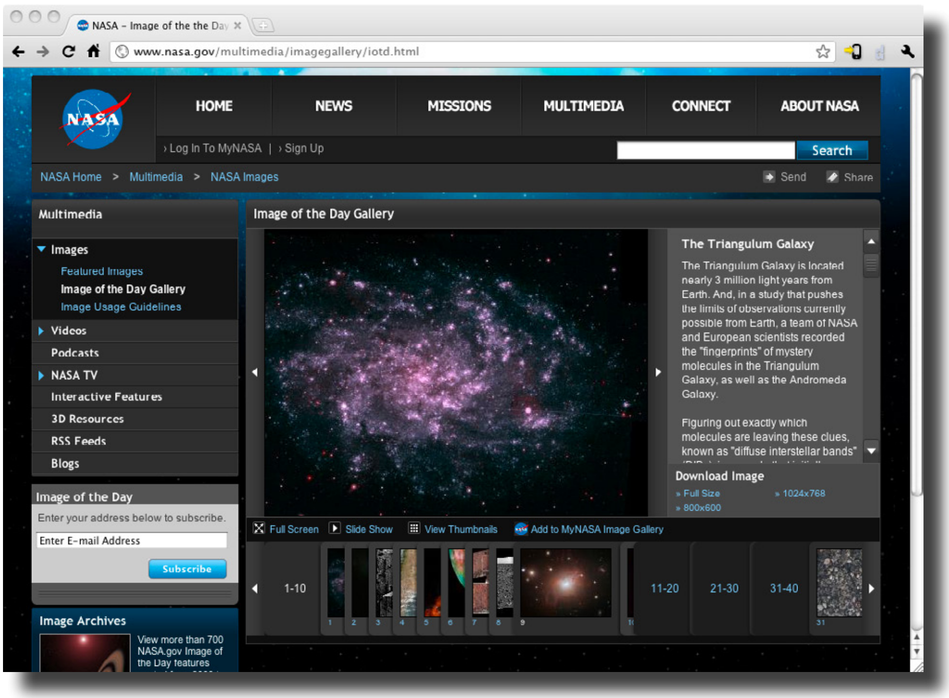
RSS feeds are everywhere! From weather and stock information to news and blogs, huge amounts of content are distributed in RSS feeds and just waiting to be used in your apps. In fact, the RSS feed publishers want you to use them! In this chapter, you'll learn how to build your own app that incorporates **content** from a public RSS feed on the Web. Along the way, you'll also learn a little more about **layouts**, **permissions,** and **debugging**.

Welcome to NASA

I really wanted a telescope, but all I can find are these binoculars. Since I can't see space, I've been checking out the NASA image of the day web site instead. It's got a cool new picture of something about space every day!



Bobby



NASA's image of the day site.

But what about phones?

The image of the day site looks pretty good on a big computer, but not so hot on a phone. It technically works, but not without a ton of scrolling and zooming. There has to be something better ...



I saw an RSS feed on NASA's site. Could you use that feed and build an Android app that reads it and displays the picture? That would be way cooler than hitting the website from my phone...



Yes! We can write an app for that!

Let's put your newly developed Android skills to use and build an app that will let Bobby see the NASA daily image on his phone. He's going to love it!

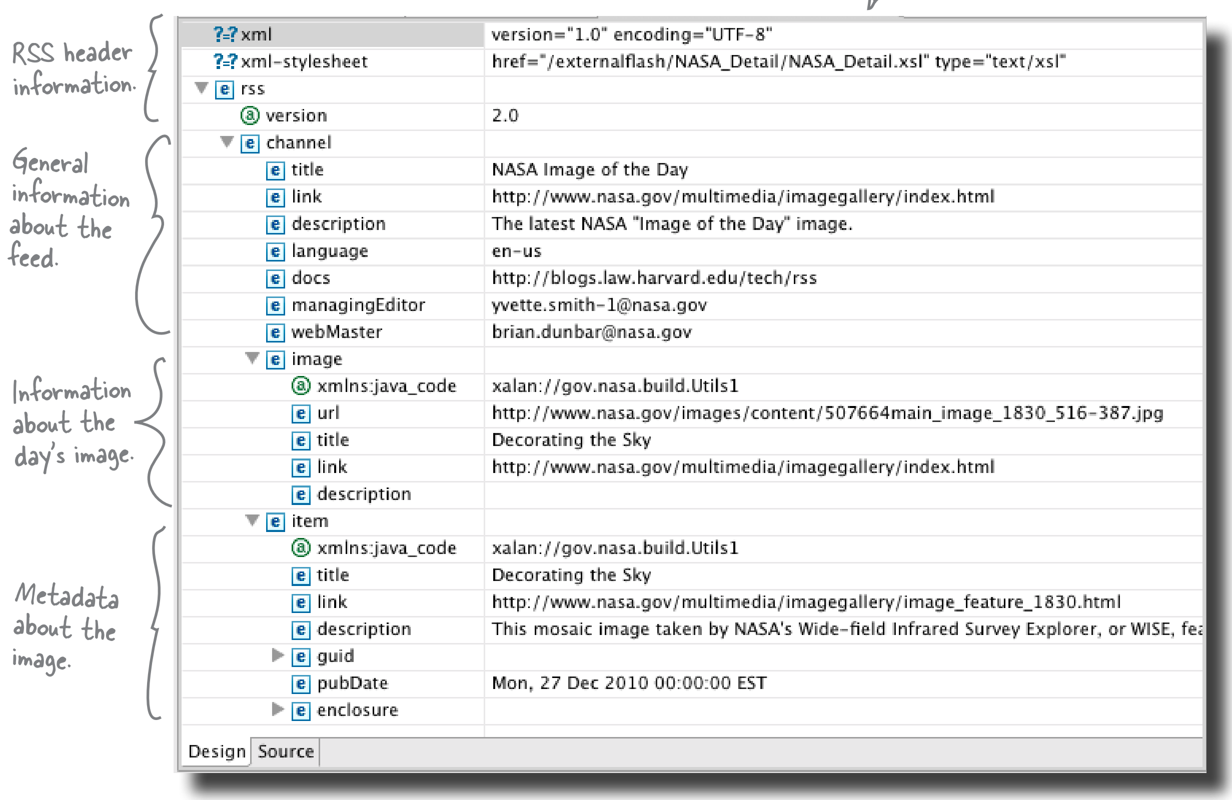
Plan out your app

Before starting on your brand-new app, take a minute to plan it out. Since you'll be building the app from he image feed from NASA, start by taking a look at the feed to get a feeling of what you have to work with.

The feed is an **RSS** (Really Simple Syndication) feed. You can find out more about RSS feeds with a quick search of the Web, but for this app, just think of it as **pure XML**.

Eclipse has a built-in XML editor that really helps to ***visualize*** the format of feeds like this. Go to **http://www.nasa.gov/rss/image_of_the_day.rss** and save the content locally on your computer as an XML file. Then you can open the XML file in Eclipse (which will automatically open the built-in XML editor) and view away!

Image of the day feed saved locally as an XML file and opened in Eclipse's XML editor.



Sharpen your pencil

There's a whole bunch of stuff in that feed! If you show it all, you're going to overload your users with information and miss the point of building a specialized mobile app for viewing the image of the day. At the same time, just showing the image would be pretty boring.

Take a look at the XML view of the feed and pick a few things you think you should show. And make sure to say why you picked it. The first one is filled in for you. Add a few more on your own.

Property to include	Why include it?
image URL	I definitely want display the image, so I'll include the image URL. This is an image of the day app, after all!

Sharpen your pencil Solution

There's a whole bunch of stuff in that feed! If you show it all, you're going to overload your users with information and miss the point of building a specialized mobile app for viewing the image of the day. At the same time, just showing the image would be pretty boring.

You were to look at the XML view of the feed, pick a few things you think you should show, and make say why you picked it.

Property to include	Why include it?
image URL	I definitely want display the image, so I'll include the image URL. This is an image of the day app, after all!
image title	The image title will help users quickly tell what the image is about.
item description	If the image is cool, users will want to read more about it. This isn't the most important information, but it's great to know.
item pubDate	NASA doesn't publish a new image every day (not on weekends, for example), so it helps to know when they did publish the image being displayed.

The XML feed doesn't include the binary image data. But using the image URL, you'll be able to doanload the image and diaply it on the screen.

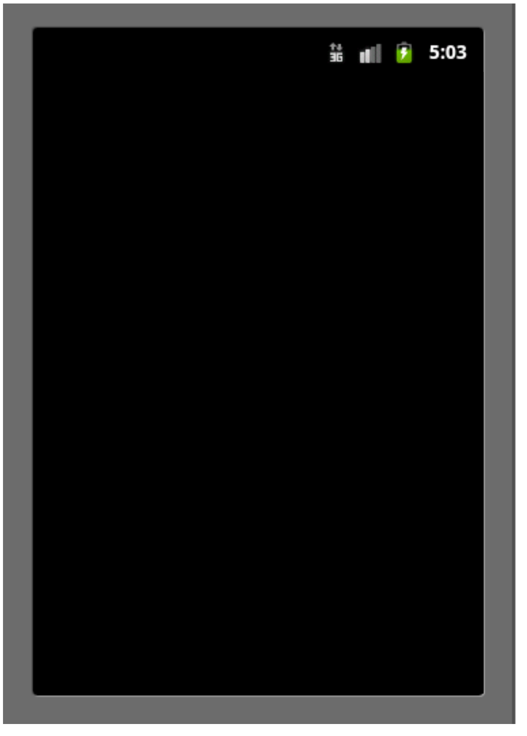
You'll need to make sure you get the correct title and description, because the example feed contains many of each. In the example feed, the image description is blank, but the item description is populated correctly.

Your answers may be slightly different and you may have picked a different field or two (and that's perfectly OK). We'll use the properties here, but there are several other perfectly good ways you could build this app.



Screen Design Magnets

To build your interface, add the `View` magnets at the bottom of this page to the screen. There is one `View` for each of the properties you picked from the RSS feed.



Put the Views on the screen here

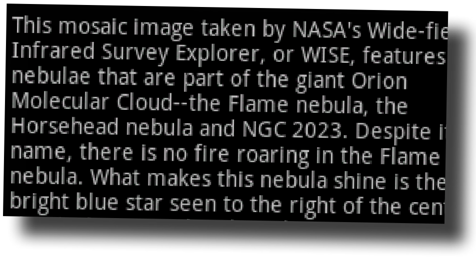
Image title in a TextView.



Item pubDate in a TextView.



Item description in a TextView.



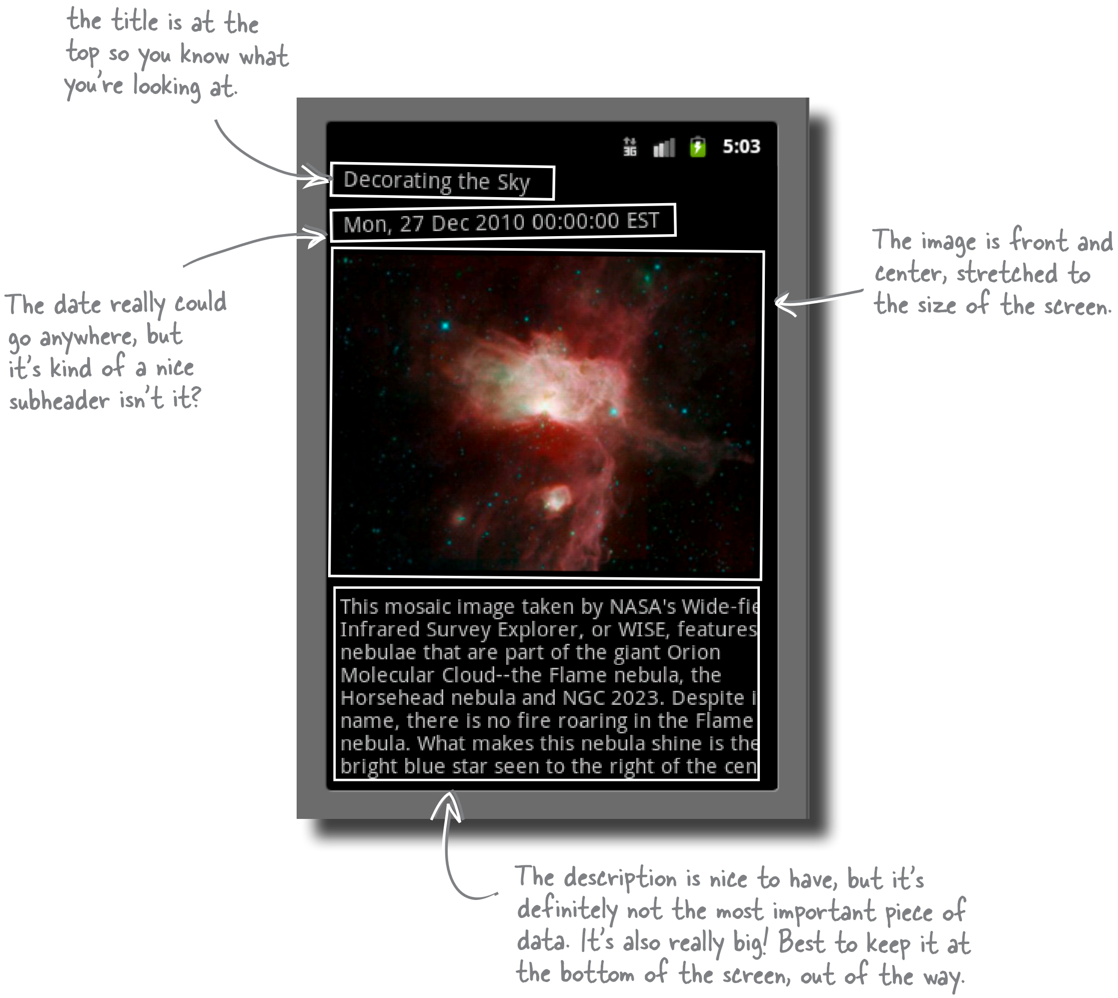
The image at the URL displayed in an ImageView (This is a new component but don't worry, you'll learn how to use it in a bit.).





Screen Design Magnet Solution

You were to add the `View` magnets at the bottom of this page to the screen to build your interface. There is one `View` for each of the properties you picked from the RSS feed.





Time to start coding!

Every good app starts with **a good plan**, and you've got one now (the selected fields from the RSS and the screen design). Now it's time to ***start coding it***.

Here is how you'll do it.

1. **Create a new project**
   You're building a new app, so start a new project. Mobile apps are small and concise, so get used to having lots of little apps (and projects) around!

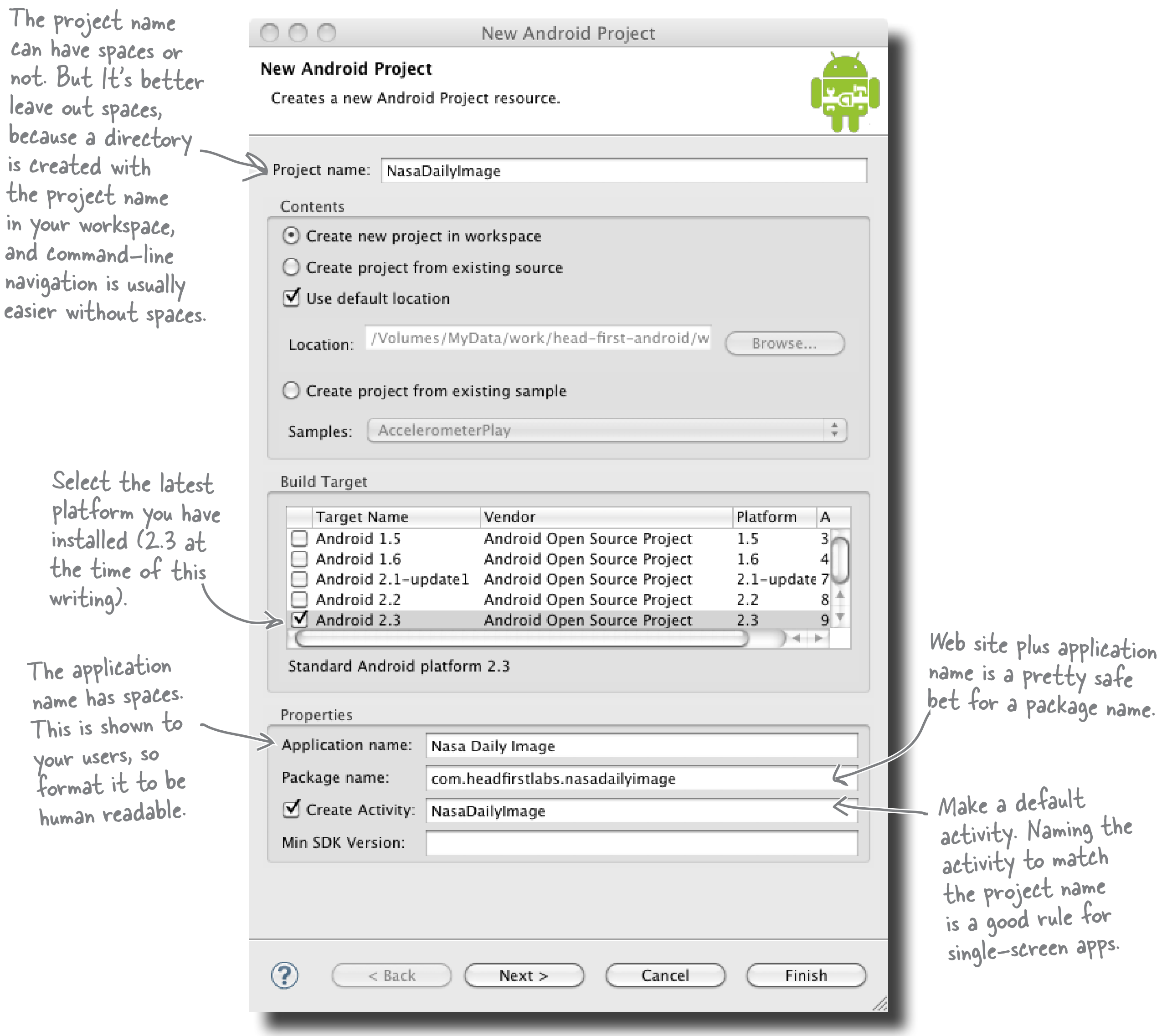
2. **Store feed information locally**
   Removing variables from development is a good thing. Store feed data locally, so you can focus on building your UI and **not** connecting to the feed.

3. **Build the UI using the stored feed data**
   You've got a design for the UI; now it's time to execute it. Create layouts, implement UI functionality, and get the app up and running!

4. **Connect the app to the XML RSS feed**
   Once the app is up and running, just plug it into the XML feed and get the live data. It really ***is*** going to be that easy. Promise!

Create a new project

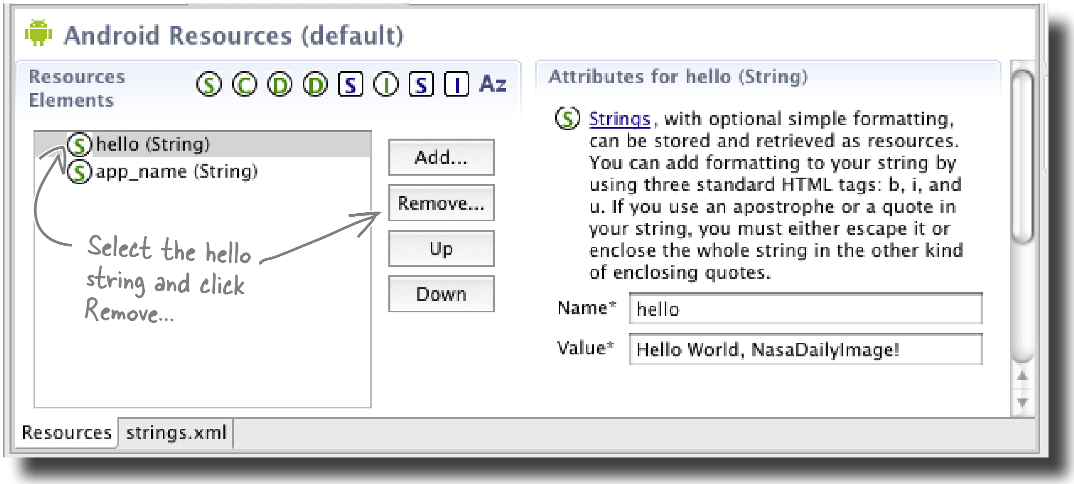
Now that you're ready to start coding, make a new Android Eclipse project. Launch the new Android project wizard in Eclipse by going to File → New → Android Project.



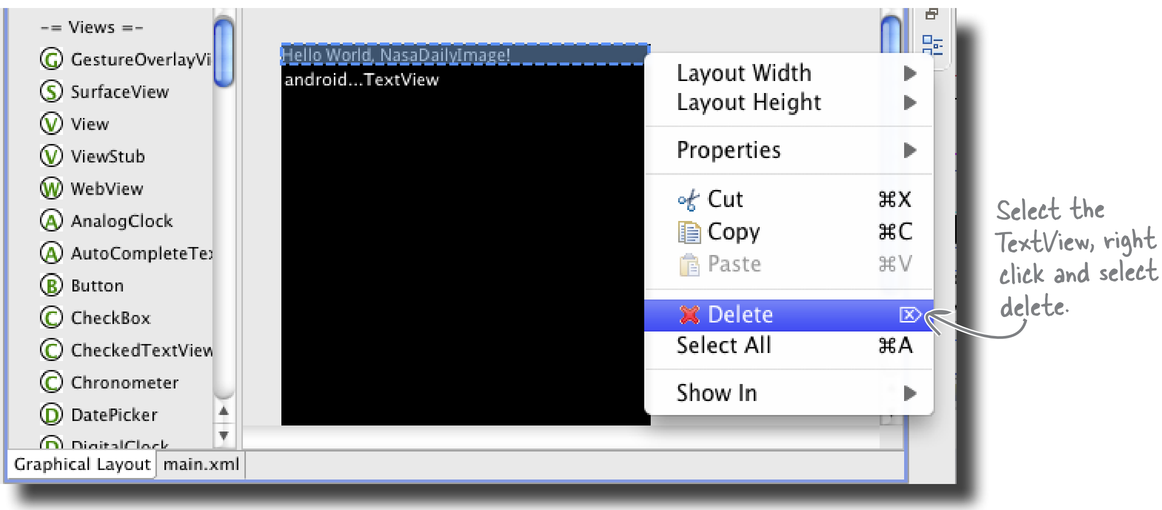
Get rid of the autogenerated 'Hello' stuff

You're not going to need the autogenerated `TextView` showing the default "Hello World, NasaDailyImage" text. So before you get going, delete the `TextView` and the the `String`.

1. Open *strings.xml* (under `res/values`) and delete the `hello String`.



2. Open *main.xml* (under `res/layout`) and delete the `hello TextView`.



3. Save your files. You now have a nice, **clean** app, without the boilerplate ***hello app*** content.

Store feed information locally

Start by saving text values as string resources. Open *strings.xml* and add three new strings for the image title, date, and description. The easiest way to do this is to copy the values directly from the sample XML feed file you saved at the beginning of the chapter.

strings.xml with the new test information added.

```
<resources>
    <string name="app_name">NASA Daily Image</string>
    <string name="test-image_title">
            Decorating the Sky
    </string>
    <string name="test-image_date">
            Mon, 27 Dec 2010 00:00:00 EST
    </string>
    <string name="test-image_description">
            This mosaic image taken by NASA\'s Wide-field
            Infrared Survey Explorer, or WISE, features
            three nebulae that are part of the giant Orion
            Molecular Cloud--the Flame nebula, the Horsehead
            nebula and NGC 2023. Despite its name, there is
```

Values from a RSS feed sample.

Watch out for places where you need to add escape characters.

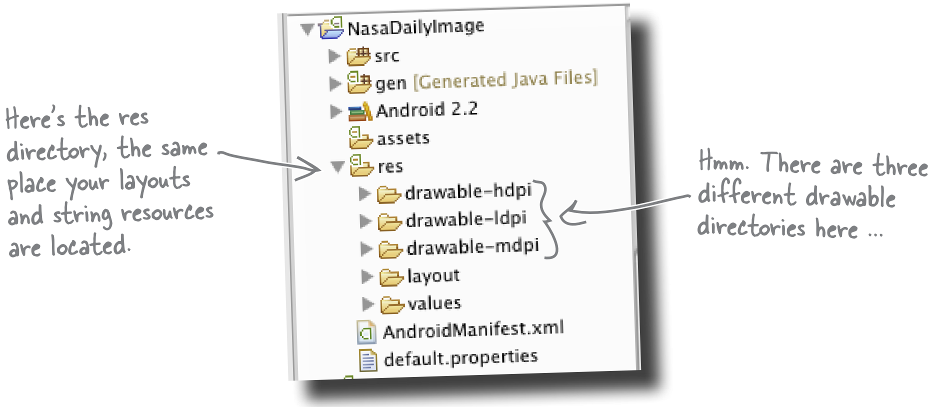


Watch out for escape characters

Some of the characters in the XML file (usually ', ", and \) need to be escaped, to let Java know they aren't control characters. Do this by preceding these characters with a \.

Save the image in your project

Images are stored in your Android project as resources in the *res* directory. Can you find a folder called *drawable* inside your project's *res* directory?



There are **three** different drawable directories under res. What gives?



Ah yes, the folders are for different screen sizes.

One of the great things about Android is how many devices it runs on... and how many devices your apps can run on! The price for that versatility is the need to support a whole bunch of different devices with a wide range of resolutions and screen sizes.

You'll learn more about supporting different screen sizes and devices later. For now, just add images to the *drawable-hdpi* directory. The default emulator will use the images in this directory.

Do this!

> Open up a browser and navigate to the URL for the image in the RSS XML file. Save the file to your project in the *res/drawable_hdmi* directory. Call it *test_image.jpg*.

Now that you have stored your data locally, let's build the layout!

Exercise

Below are magnets with the XML layout declarations for the `Views` in your layout along with the the `Views` they represent. Drag the `View` XML magnets onto the layout on the next page of the exercise. This will complete the layout for the app.

View visuals here, just for reference.

View XML declaration magnets

Image



```
<ImageView
      android:id="@+id/imageDisplay"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:src="@drawable/test_image"/>
```

Decorating the Sky

Title

```
<TextView
      android:id="@+id/imageTitle"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/test_image_title"/>
```

This mosaic image taken by NASA's Wide-field Infrared Survey Explorer, or WISE, features three nebulae that are part of the giant Orion Molecular Cloud--the Flame nebula, the Horsehead nebula and NGC 2023. Despite its name, there is no fire roaring in the Flame nebula. What makes this nebula shine is the bright blue star seen to the right of the central

```
<TextView
      android:id="@+id/imageDescription"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/test_image_description"/>
```

Description

Mon, 27 Dec 2010 00:00:00 EST

Date

```
<TextView
      android:id="@+id/imageDate"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/test_image_date"/>
```

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
       android:orientation="vertical"
       android:layout_width="fill_parent"
       android:layout_height="fill_parent" >
```

Put the widget magnets here to complete the layout. You're using LinearLayout, so you just need to arrange them with the component at the top of the screen as the first in the layout and continuing down.

```
</LinearLayout>
```

Exercise Solution

Below are magnets with the XML layout declarations for the `Views`. You were to arrange the the `View` XML magnets to complete the layout for the app.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" >

    <TextView
            android:id="@+id/imageTitle"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/test_image_title"/>

    <TextView
            android:id="@+id/imageDate"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/test_image_date"/>

    <ImageView
            android:id="@+id/imageDisplay"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:src="@drawable/test_image"/>

    <TextView
            android:id="@+id/imageDescription"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/test_image_description"/>

</LinearLayout>
```

Title

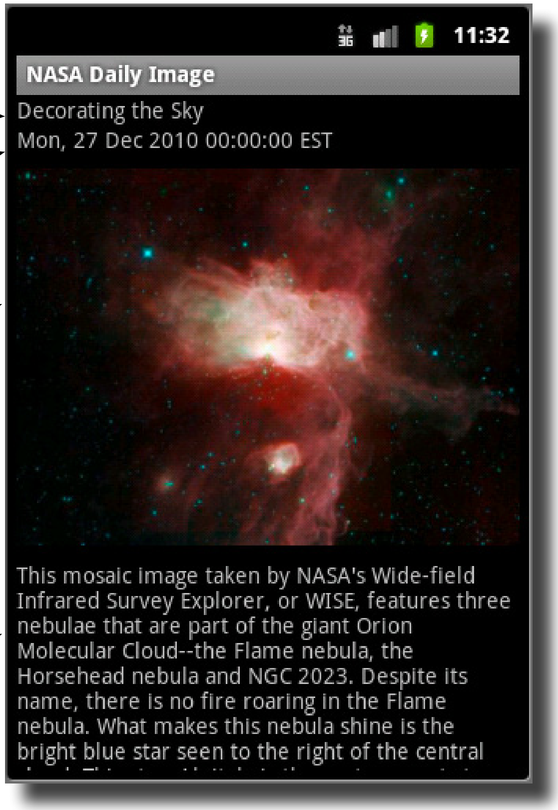
Date

Image

Description

Test Drive

Run the app by selecting the project in the Exclipse explorer view and selecting run. You'll have to select Android Application in the "Run as" pop-up that displays.



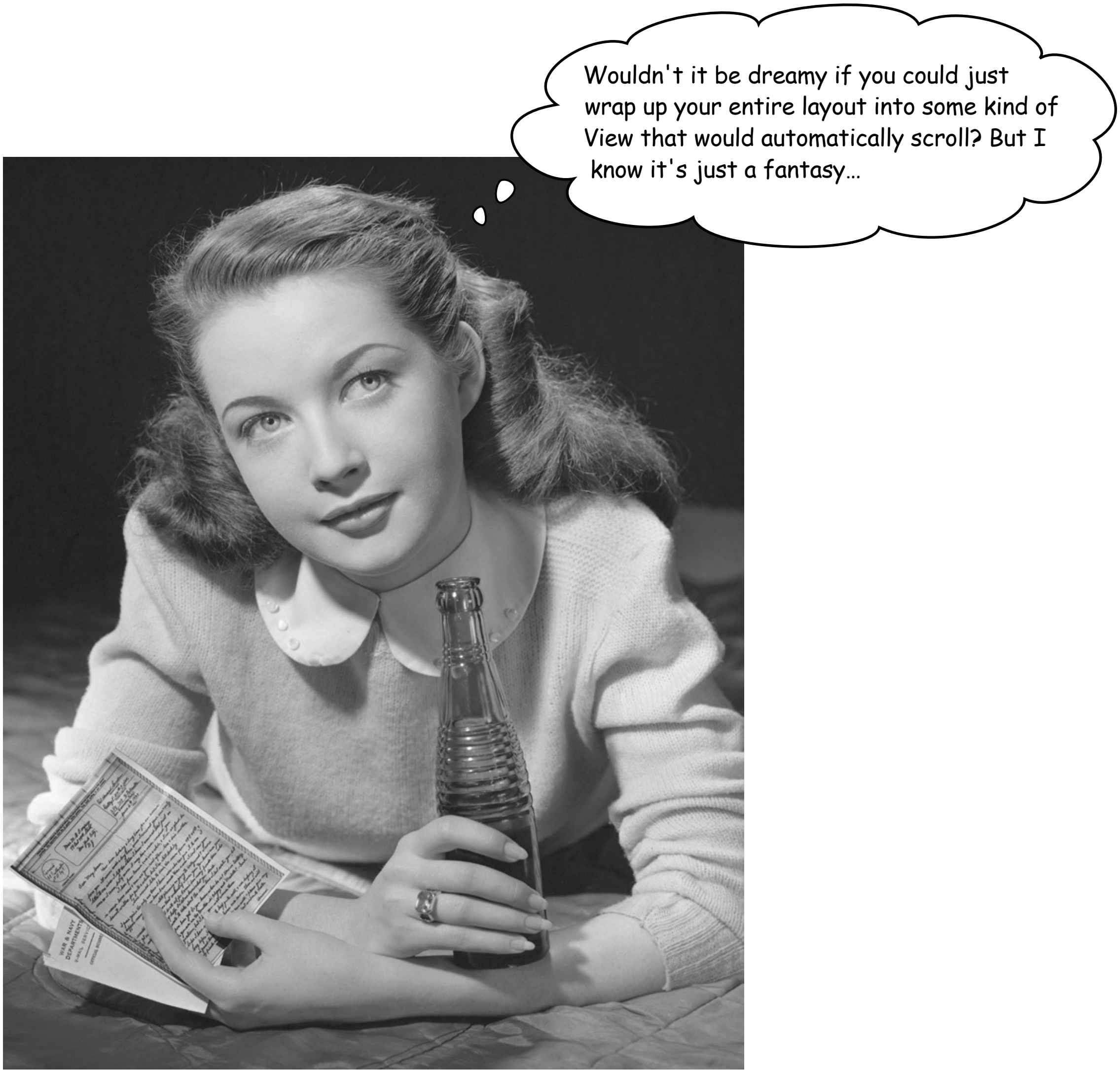
Nice! The screen is looking good!

The running screen matches your design. Excellent work.



Actually, scrolling *would* be a good idea!

You never know how long the description might be. NASA could throw a whole book in there, for all we know! After all, they are in control of the feed. The best we can do is make our app ***visually scalable***. A good way to do that is just to make the entire screen scroll.


Wouldn't it be dreamy if you could just wrap up your entire layout into some kind of View that would automatically scroll? But I know it's just a fantasy...

Use ScrollView to show more content

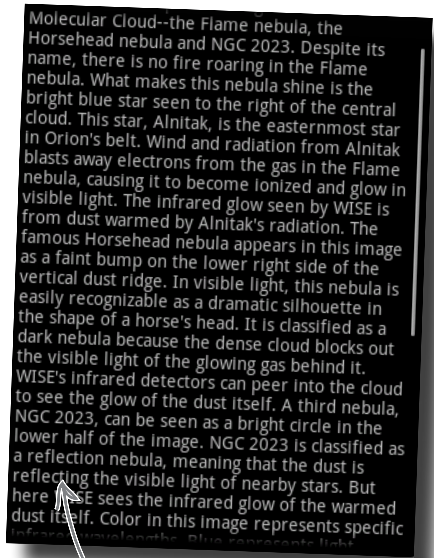
`ScrollView` is a `View` you can add to your screens to make content scroll. `ScrollView` is a `ViewGroup` (Android's name for layout manager). Use `ScrollView` by adding a child component to it, and the `ScrollView` will automatically scroll.

This ScrollView wraps the description TextView ...

```
<ScrollView
       android:layout_width="fill_parent"
       android:layout_height="fill_parent" >
       <TextView
         android:text="@string/image_description"
         android:layout_width="fill_parent"
         android:layout_height="wrap_content" />
</ScrollView>
```

The description TextView is big enough to overfill the screen by itself!

... and here's that TextView scrolling inside the ScrollView!



LOTS of scrolling text!

How much should scroll?

You can put one or more of the existing `Views` into the `ScrollView`. Any `Views` you add to the `ScrollView` will scroll, and the views not in the scrollview won't. Since your goal is visual scalability, just make the ***entire layout scroll***. This way, you can be gauranteed to have a **scalable** UI, even if unepxected information comes through the feed (like a *really* long title, for example).

One catch using `ScrollView` is that it can have only a single child `View`. In the example on this page, the `TestView` is added directly as a child to the `ScrollView`. But for the *whole* screen to scroll, you need multiple `Views` to scroll. The solution is to add a complete `LinearLayout` (with multiple child `Views`) as the `ScrollView`'s child.

Sharpen your pencil

Add and amend the following code to use the ScrollView to make the entire screen scroll. You'll need to make the ScrollView the main layout. And since the ScrollView can hold only one View, you need to add the entire LinearLayout as the one ScrollView child View.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" >

<TextView
        android:id="@+id/imageTitle"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/test_image_title"/>
<TextView
        android:id="@+id/imageDate"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/test_image_date"/>
<ImageView
        android:id="@+id/imageDisplay"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/test_image"/>
<TextView
        android:id="@+id/imageDescription"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/test_image_description"/>

</LinearLayout>
```

Quick Tip: This needs to be in the root layout. If you add this layout to a ScrollView, you'll need to move this to the ScrollView.

Wrap this entire layout in a ScrollView.

Sharpen your pencil Solution

You were to used the `ScrollView` to make the entire screen scroll. You needed to make the `ScrollView` the main layout. And since the `ScrollView` can hold only one `View`, you should have added the entire `LinearLayout` as the one `ScrollView` child `View`.

```
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent" >
    <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" >
    <TextView
        android:id="@+id/imageTitle"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/test_image_title"/>
    <TextView
        android:id="@+id/imageDate"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/test_image_date"/>
    <ImageView
        android:id="@+id/imageDisplay"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/test_image"/>
    <TextView
        android:id="@+id/imageDescription"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/test_image_description"/>
    </LinearLayout>
</ScrollView>
```

Did you remember to move the xmlns:android attribute from the LinearLayout to the ScrollView (the root view)

Beginning of the ScrollView

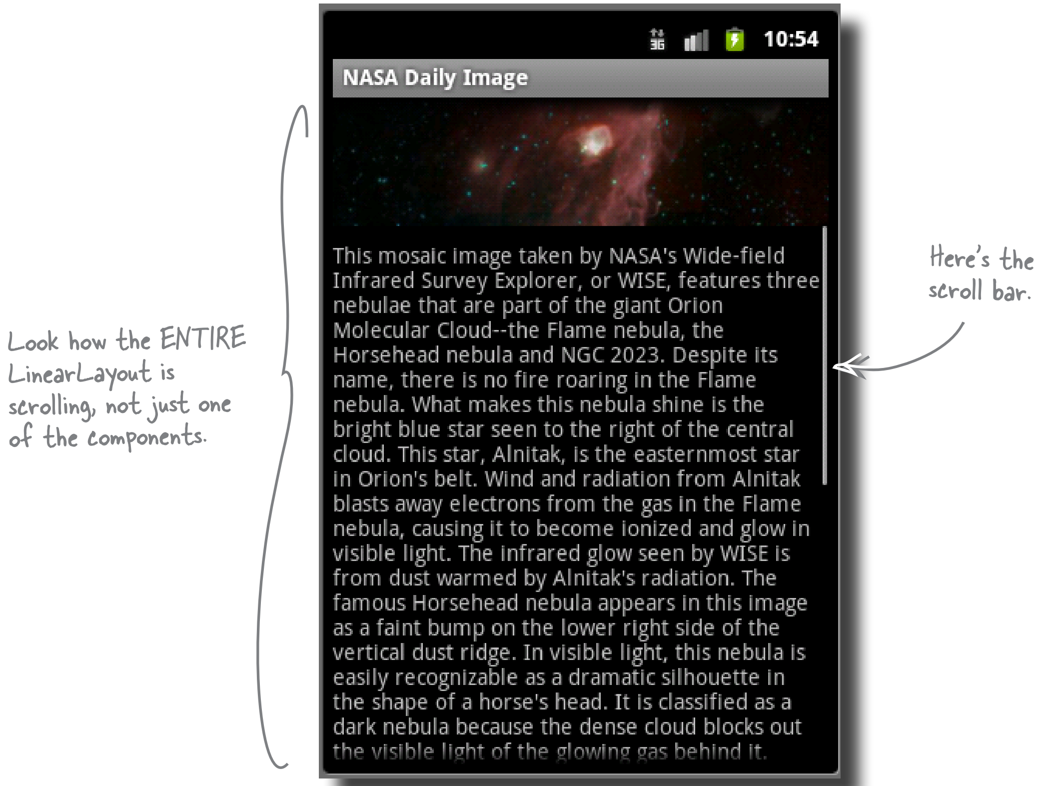
The complete non-scrolling layout

The inner widgets remain untouched inside the LinearLayout.

end of the ScrollView

Test Drive

Run your app to check the scrolling you just added. You should see the entire screen scrolling.



Everything is scrolling as expected.

The scrolling is working properly. See how the ***entire screen*** content scrolls up and down together? That's because you added the entire `LinearLayout` as the child to the `ScrollView`.

Let's show it to Bobby and see what he thinks!



Oops! Almost forgot about the actual feed.

Things are going really well with the design and layout. The screen ***looks*** like you want. Now it's time to make it ***work*** the way you want... parsing the feed data in real time.

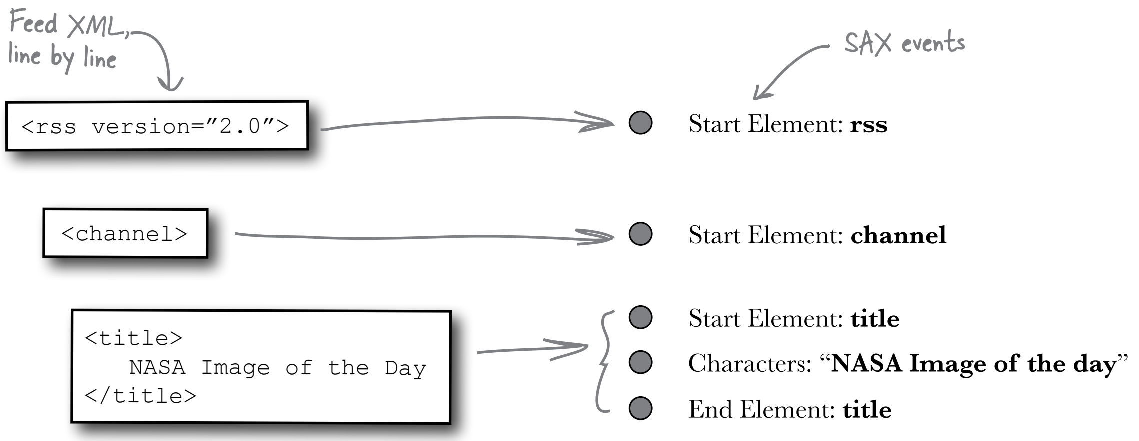
Choose a parser

There are plenty of XML parsers out there, and Android has built-in support for three of them: **DOM** (Document Object Model), **SAX** (Simple API for XML), and **XMLPULL**. They each take a different approach to parsing the XML and each has benefits and drawbacks. We're going to skip the big XML parser smackdown here (don't worry, though, you can find plenty on the Web) and just pick one.

Let's keep it simple and start with SAX.

SAX Parsing Up Close

SAX works by firing events while parsing the XML. There is no random access with SAX. The parser begins at the beginning of the XML, fires appropriate messages, and exits. Here's a quick sample of a few events that get fired in the first three lines of the NASA image feed.



The parser for the NASA feed will need to listen for the SAX start element messages for the fields in the app (the title, image URL, description, and date) and cache the values. *That's it!*

Let's review some Ready Bake parser code to keep you moving!

Ready Bake Code

SAX-based feed parsers look pretty much the same. Now that you understand how the SAX parser ***conceptually*** works, here is a parser packaged up as Ready Bake code that you can just drop into your app. Don't worry about understanding everything; just add it to your project. But feel free to explore it!

```
public class IotdHandler extends DefaultHandler {
        private String url = “http://www.nasa.gov/rss/image_of_the_day.rss”;
        private boolean inUrl = false;
        private boolean inTitle = false;
        private boolean inDescription = false;
        private boolean inItem = false;
        private boolean inDate = false;
        private Bitmap image = null;
        private String title = null;
        private StringBuffer description = new StringBuffer();
        private String date = null;

        public void processFeed() {
                try {
                        SAXParserFactory factory =
                                SAXParserFactory.newInstance();
                        SAXParser parser = factory.newSAXParser();
                        XMLReader reader = parser.getXMLReader();
                        reader.setContentHandler(this);
                        InputStream inputStream = new URL(url).openStream();
                        reader.parse(new InputSource(inputStream));
                } catch (Exception e) {
        }
        private Bitmap getBitmap(String url) {
                try {
                        HttpURLConnection connection =
                                (HttpURLConnection)new URL(url).openConnection();
                        connection.setDoInput(true);
                        connection.connect();
                        InputStream input = connection.getInputStream();
                        Bitmap bitmap = BitmapFactory.decodeStream(input);
                        input.close();
                        return bitmap;
                } catch (IOException ioe) { return null; }
        }
```

Since the events get called separately (like starting elements and their contents), keep track of what element you're in ...

Configuring the reader and parser.

Make an input stream from the feed URL.

Start the parsing!

```
        public void startElement(String uri, String localName, String qName,
                Attributes attributes) throws SAXException {
            if (localName.equals("url")) { inUrl = true; }
            else { inUrl = false; }

            if (localName.startsWith("item")) { inItem = true; }
            else if (inItem) {
                if (localName.equals("title")) { inTitle = true; }
                else { inTitle = false; }

                if (localName.equals("description")) { inDescription = true; }
                else { inDescription = false; }

                if (localName.equals("pubDate")) { inDate = true; }
                else { inDate = false; }
            }
        }

        public void characters(char ch[], int start, int length) {
            String chars = new String(ch).substring(start, start + length);
            if (inUrl && url == null) { image = getBitmap(chars); }
            if (inTitle && title == null) { title = chars; }
            if (inDescription) { description.append(chars); }
            if (inDate && date == null) { date = chars; }
        }
    }

    public String getImage() { return image; }
    public String getTitle() { return title; }
    public StringBuffer getDescription() { return description; }
    public String getDate() { return date; }
```

.... and if you're in an element that you are interested in, cache the characters.

Here are a few accessors. so you can get the cached variables back from the parser...

Do this!

Download the `IotdHandler` code from the *Head First Android Development* site and add it to your project.

Connect the handler to the activity

Now that you've added the feed parser code to your project, you need to use it in your activity. Start by instantiating the handler in your Activities `onCreate` method.

```
public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        IotdHandler handler = new IotdHandler();
        handler.processFeed();
}
```

Create the handler ... (annotation on `IotdHandler handler = new IotdHandler();`)

and start parsing ... (annotation on `handler.processFeed();`)

The app's not going to work with the parser yet. You're parsing the feed, but you're not setting the values cached in the feed on the Views.



True, the values are cached in the handler, but never displayed.

Let's make a method called `resetDisplay` that will set all of the view data on screen. Then you can call that method in `onCreate()` after`processFeed()` returns.



Code Magnets

Complete the `resetDisplay()` method below by retrieving references to the on-screen `Views` (using `findViewById`) and setting the values on those `Views` with the values passed in. Once this method is complete, you can use it to pass in the values from the feed.

```
private void resetDisplay(String title, String date,
       String imageUrl, String description) {
```

Get a reference to each on screen View. Then set the values on those Views to the cached values from the parser.

```
}
```

Here are your magnets.

```
(TextView)findViewById(R.id.imageDate);
(TextView)findViewById(R.id.imageTitle);
titleView.setText(title);
TextView descriptionView =
(ImageView)findViewById(R.id.imageDisplay);
(TextView)findViewById(R.id.imageDescription);
dateView.setText(date);
ImageView imageView =
imageView.setImageBitmap(image);
descriptionView.setText(description);
TextView dateView =
TextView titleView =
```



Code Magnets Solution

You were to complete the `resetDisplay()` method below by retrieving references to the on screen `Views` (using `findViewById`) and setting the values on those `Views` with the values passed in. With this method complete, you can use it to pass in the values from the feed.

```
private void resetDisplay(String title, String date,
        String imageUrl, String description) {

    TextView titleView = (TextView)findViewById(R.id.imageTitle);
    titleView.setText(title);
```

Get a reference to the title view and set the text to the cached value from the handler.

```
    TextView dateView = (TextView)findViewById(R.id.imageDate);
    dateView.setText(date);
```

Same deal with date View: get the View reference and set the text to the value from the parser.

Get a reference to the ImageView.

```
    ImageView imageView = (ImageView)findViewById(R.id.imageDisplay);
    imageView.setImageBitmap(image);
```

Use the image from the feed parser and set it on the ImageView.

```
    TextView descriptionView = (TextView)findViewById(R.id.imageDescription);
    descriptionView.setText(description);

}
```

Finish up by getting the description View reference and setting the text with the cached description value.

Now you can finish connecting the handler in the `onCreate()` method. Add a call to `resetDisplay()` after `handler.processFeed()`. This will take the cached values in the parser and set them in the `Views` screen.

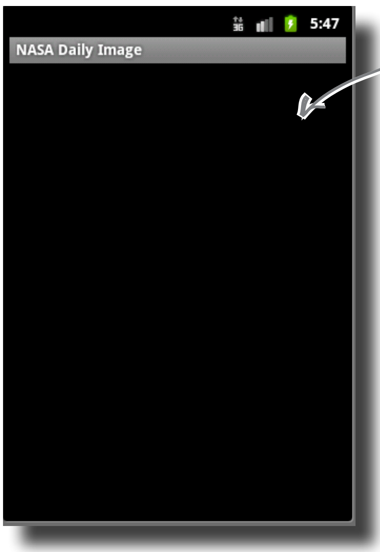
```
resetDisplay(iotdHandler.getTitle(), iotdHandler.getDate(),
    iotdHandler.getImage(), iotdHandler.getDescription());
```

The resetDisplay method is a helper method you're about to write to populate the fields on screen with the parsed data.



Test Drive

Everything is plugged in with the parser. The parser is integrated with the activity, and the results from the parsing are displayed on the screen. You should be good to go. Go ahead and run the app.



Hmm, a blank screem...

Uh oh! The screen is gone!



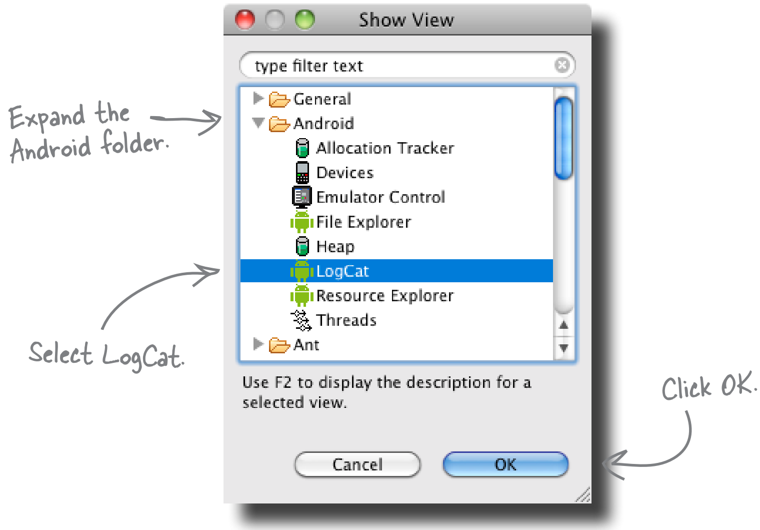
Brain Barbell

Clearly, something broke along the way. What broke? Where would you look to find out what's broken?

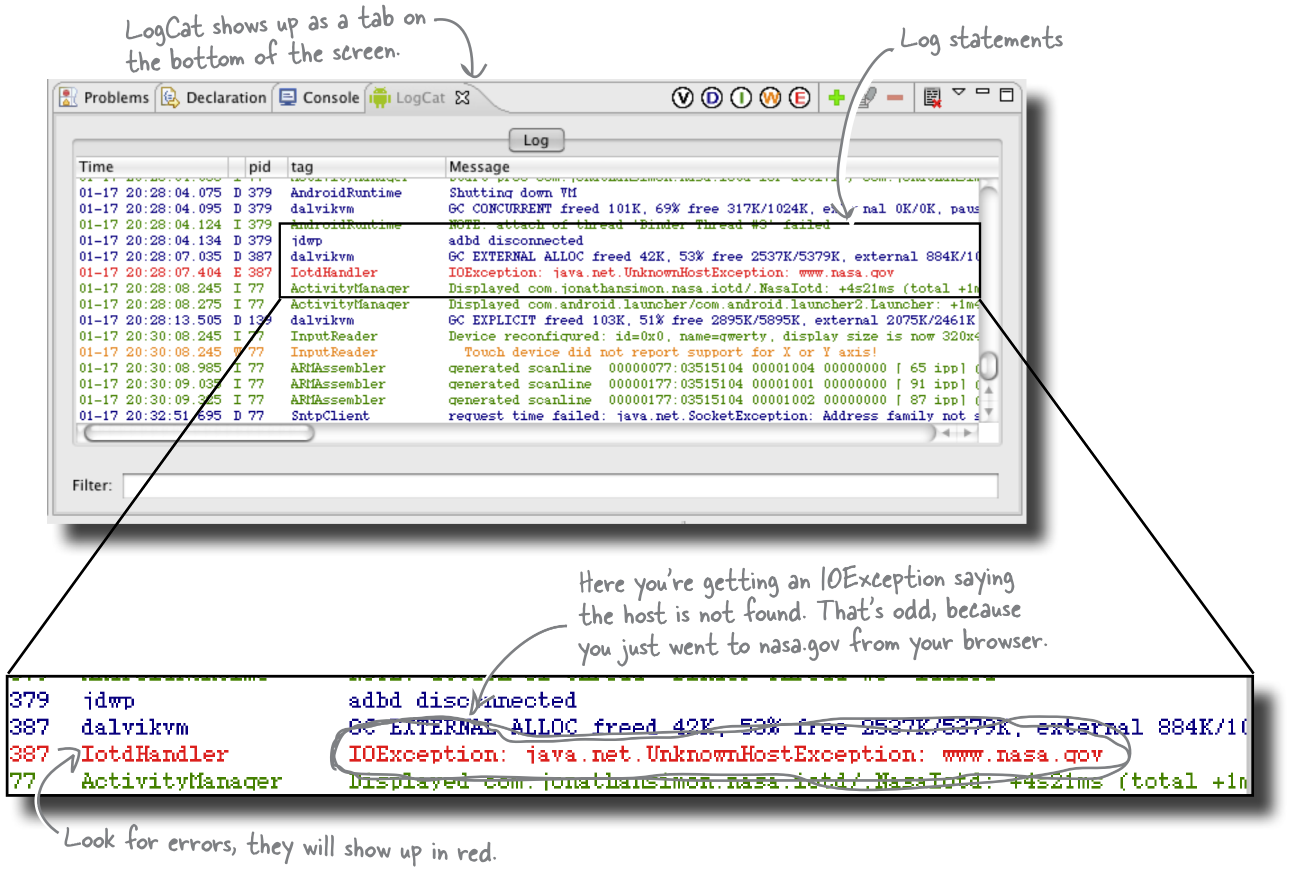
Find errors with LogCat

It's OK; errors happen! The important thing is knowing where to go to find out what's happening with your application, so you can fix things when they break. Android uses a built-in logging mechanism that outputs to a screen included in the Android Development Tools (ADT) called *LogCat*.

Open LogCat by going to Window → Show View → Other, which will bring up the Eclipse Show View dialog. Expand the Android folder, select LogCat, and press OK.



After you click OK, you'll see the new LogCat view in your Eclipse workspace.



Use permissions to gain restricted access

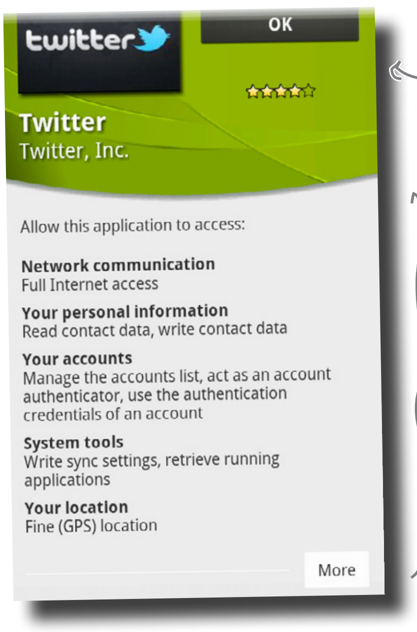
The `UnknownHostException` is thrown here because you need ***permission*** to access the Internet.

With all the cool stuff you can do with Android devices, it's hard to remember that they are **mobile devices**. And because of this, Android is built to be super careful about making sure each app has rights only to the system resources it ***absolutely needs***. The only way for your app to get those permissions is to request them.

How do permissions work?

You can specity the permissions your app needs using a group of permission constants in *AndroidManifest.xml*. When users install your app from the Android market, they are prompted with a list of permissions that your app requsts. If they agree, they accept the permissions and the app installs.

As an example, let's take a look at the Android market install page for the official Twitter app.



The market install page for the Twitter app

All of the permissions the app requests

The permissions on the market page are generated from these constants.

```
android.permission.INTERNET

android.permission.READ_CONTACTS
android.permission.WRITE_CONTACTS

android.permission.MANAGE_ACCOUNTS
android.permission.AUTHENTICATE_ACCOUNTS
android.permission.USE_CREDENTIALS

android.permission.WRITE_SYNC_SETTINGS
android.permission.GET_TASKS

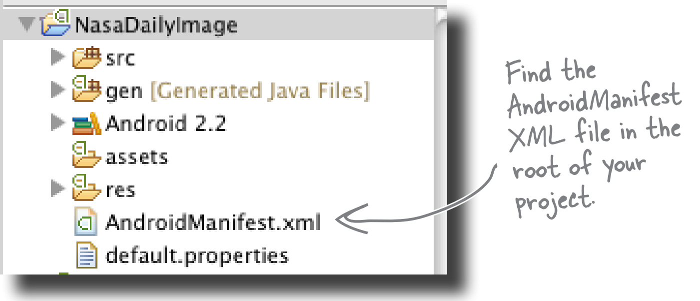
android.permission.ACCESS_FINE_LOCATION
```

Enough about Twitter. Let's add permission to your app!

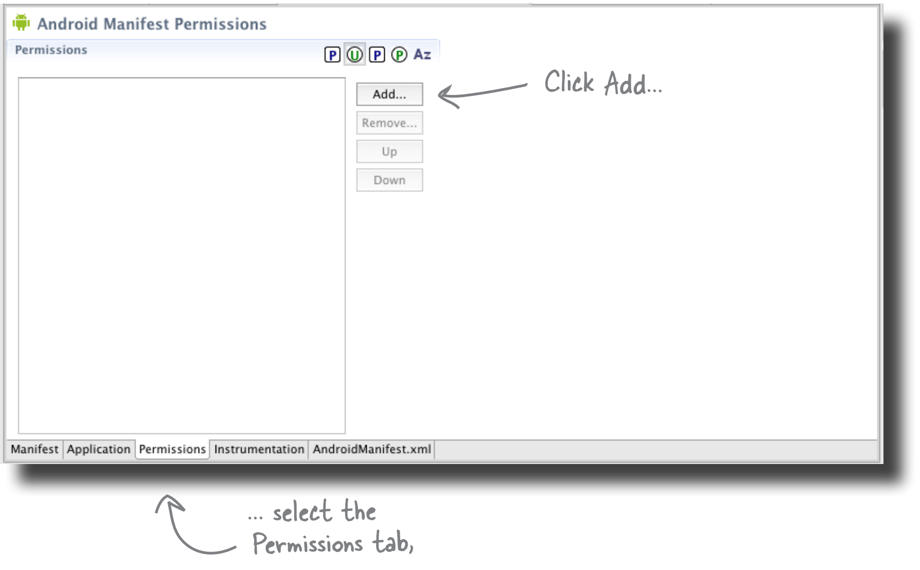
Add a permission to access the internet

The Twitter app had a lot of permissions, but your app just needs permission to access the Internet. Follow these instructions to add the Internet access permission.

1 **Open AndroidManifest.xml**
The *AndroidManifest* file is generated by the new app wizard. You can find it in the root of your project. Double-click the file to open it.

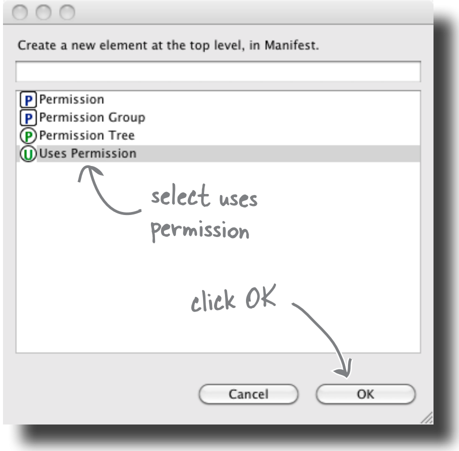


2 **Add a new permission to the manifest**
Just like all of the other Eclipse XML editors you've been working with, there's a custom editor for *AndroidManifest* file. Click on the Permissions tab and press the Add button to add a new permission.



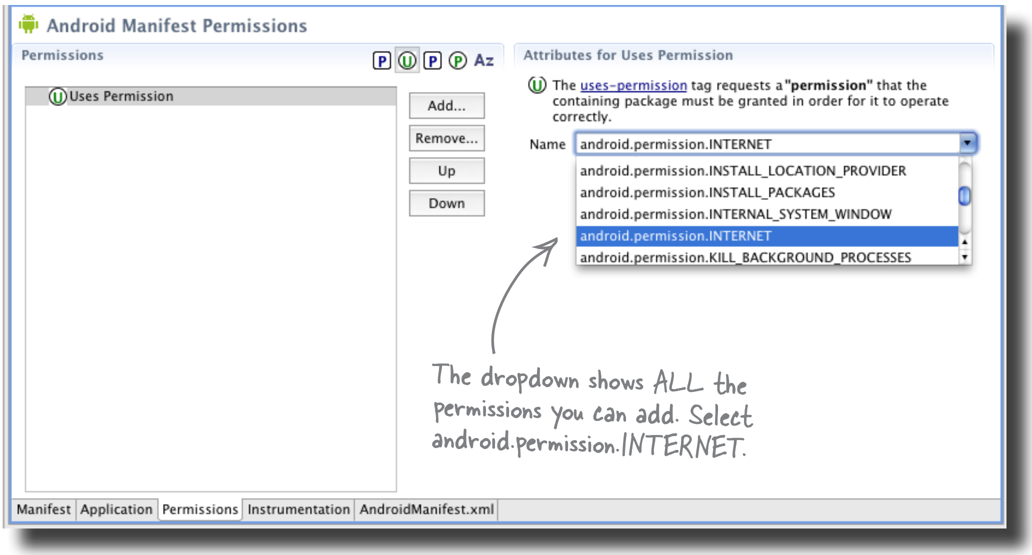
3 Select the permission type

When the dialog opens, select Uses Permission and click OK. This tells Android that you want to use a permission in your application.



4 Select the permission

There are a bunch of different permissions that you can ad to your application. Since you're accessing the Internet to get the feed and the image, select the `android.permission.INTERNET` permission.



To apply the changes, save the file when you're done.



Tonight's talk: **Permissions**

Android App:

What, seriously? I have to ask permission to do everything? Don't you trust me at all? This is ridiculous!

Android Operating System:

No, it isn't ridiculous. I just need to be *really* careful about what I let you do unsupervised.

Android App:

Unsupervised?!? Look, I'm not a child!

Android Operating System:

Well, listen, my user (who is also your user I might add) expects us all to work together to keep the whole phone ***secure***. We can't allow any viruses, unauthorized data access, unecessary Internet access, or other security no-nos to spoil their experience. Then we all lose!

Android App:

OK, well I kind of see that. But really, I have to tell you everything I do? Like *everything*? That's lame!

Android Operating System:

Sorry, but you do. That way, I can tell our user what you're planning on doing and they can decide if they will let you do it.

Android App:

Why can't I just ask them myself?

Android Operating System:

How can I trust that if the user says no to you you'll actually listen? You wouldn't even listen to me if I couldn't kill your process!

Android App:

Hey man, that's low.

Android Operating System:

Well would you?

Android App:

You're right, I probably wouldn't. BUT ...

Android Operating System:

I rest my case!

Android App:

Mffft! Well, I suppose I don't really have a choice, do I?

Android Operating System:

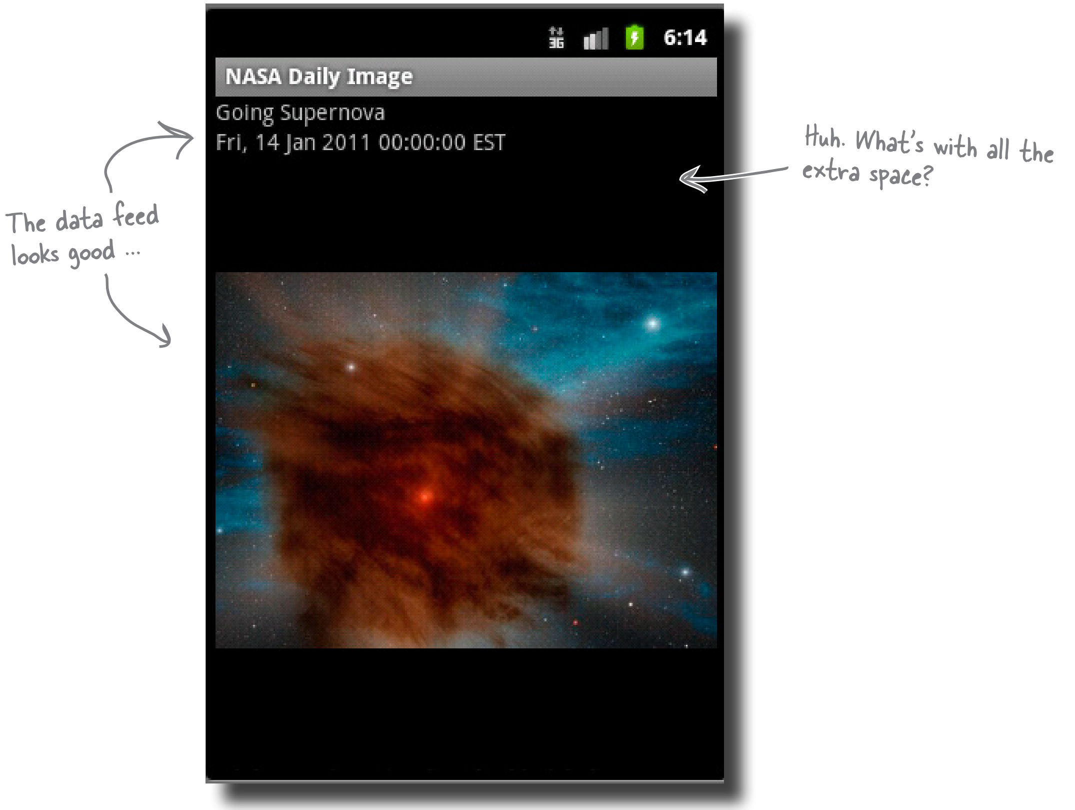
Nope! You don't have a choice. My way, or the highway, buddy.

Android App:

Harsh.

Test Drive

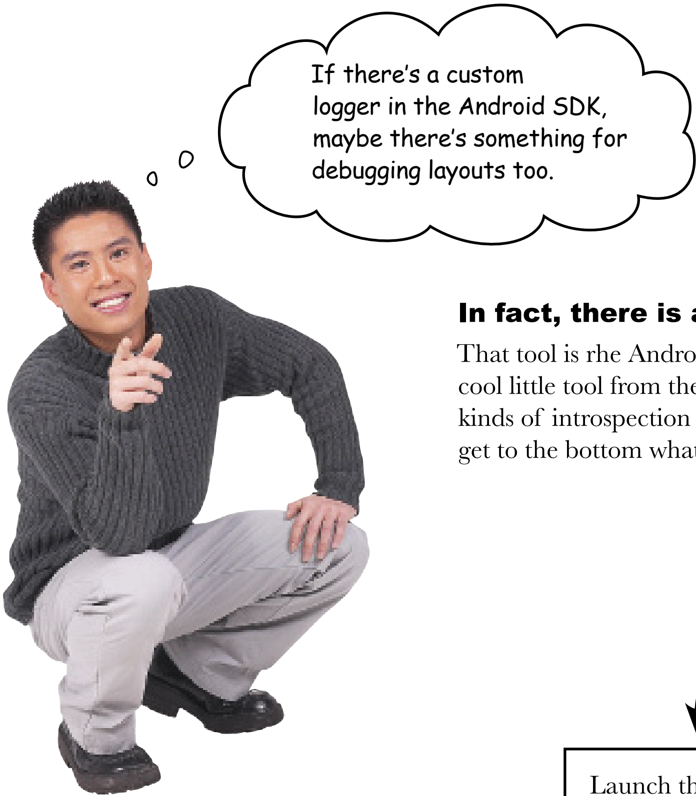
Now that the permissions are properly set, the app should run correctly, parsing the feed and displaying everything on the screen. Go ahead and run your app!



Better, but not done yet!

The feed is working (***fantastic!***), and fresh data is bing displayed on the screen. This is all great, but something is going wrong with the formatting.

How do you find out what's wrong?



In fact, there is a built-in tool.

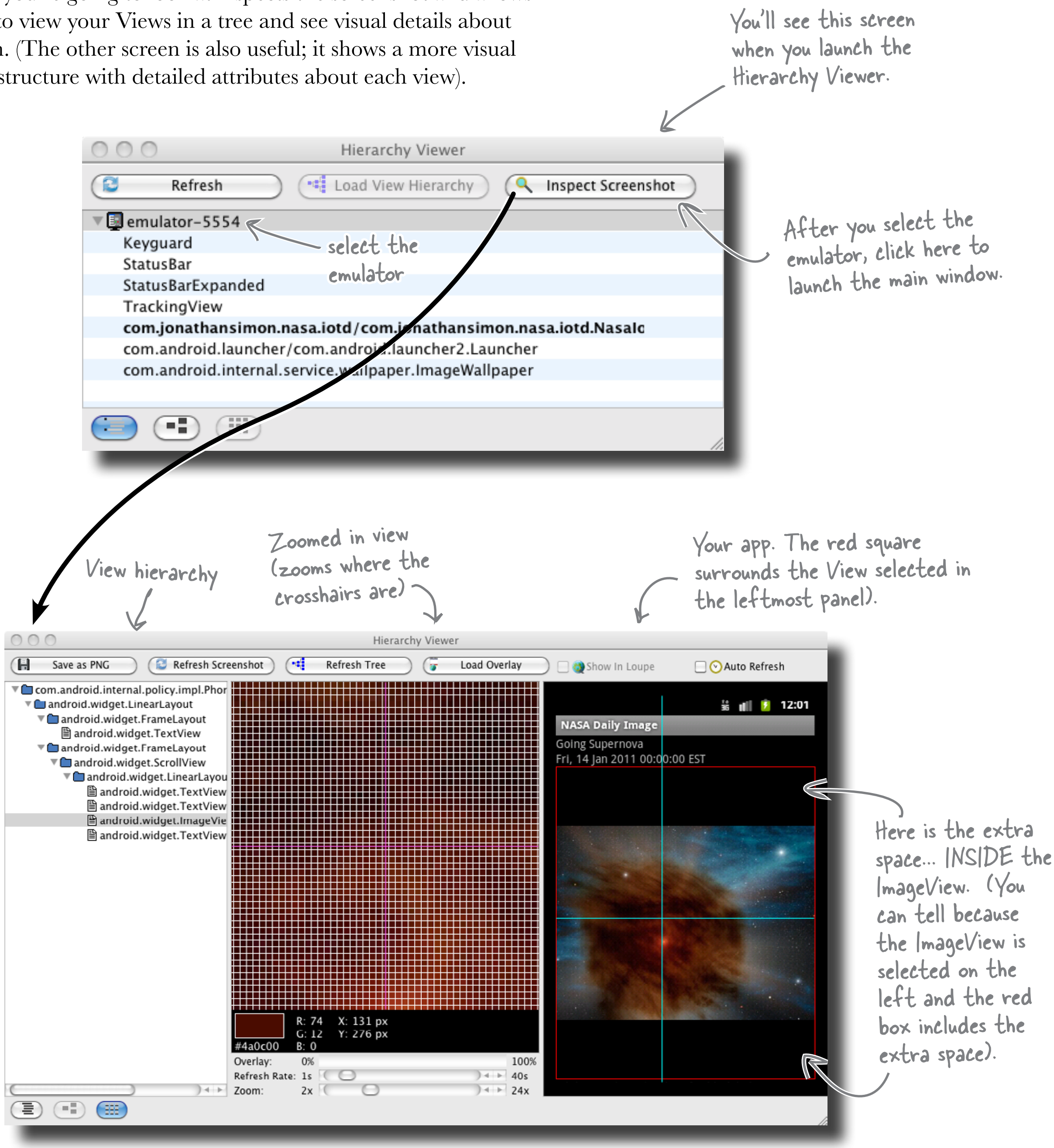
That tool is rhe Android **Hierarchy Viewer**. This cool little tool from the **Android SDK** lets you do all kinds of introspection on your layouts and Views to get to the bottom what's going on.

Do this!

Launch the Hierarchy Viewer by opening a terminal, going to your *<SDK>/tools* directory, and executing **`hierarchyviewer`** at the command line.

Find layout problems with HierarchyViewer

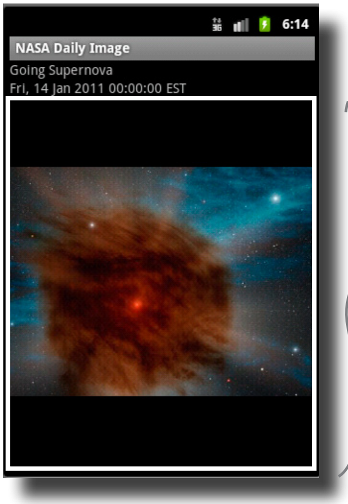
When you launch the **Hierarchy Viewer**, the first thing you'll see is the selection screen below. There are two main views; the view you're going to look at inspects the screenshot and allows you to view your Views in a tree and see visual details about them. (The other screen is also useful; it shows a more visual tree structure with detailed attributes about each view).



Set the adjustViewBounds property

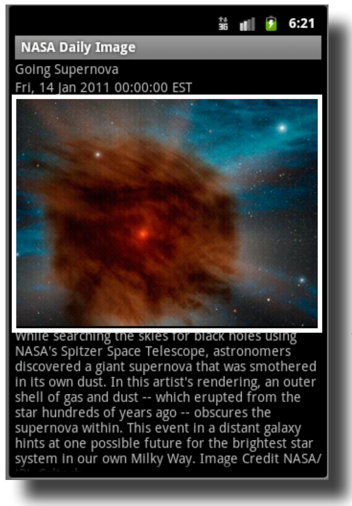
You can see from the Hierarchy Viewer that the `ImageView` is too big. But why? The cause is actually that the apsect ratio is not preserved when the `Bitmap` from the Web is displayed. The **aspect ratio** is what keeps the width to height ***proptionally*** the same when you resize an image, and the image is being resized by the internal layout code to fill the screen width.

`adjustViewBounds =` **`false`**



Without keeping the aspect ratio the same, the image stretches and takes up too much space.

`adjustViewBounds =` **`true`**



When set to true, the image stretches to the edges of the screen, and sets a height proprtional to that width.

If you set the `adjustViewBounds` property to **true** in your layout XML, the extra space will go away.

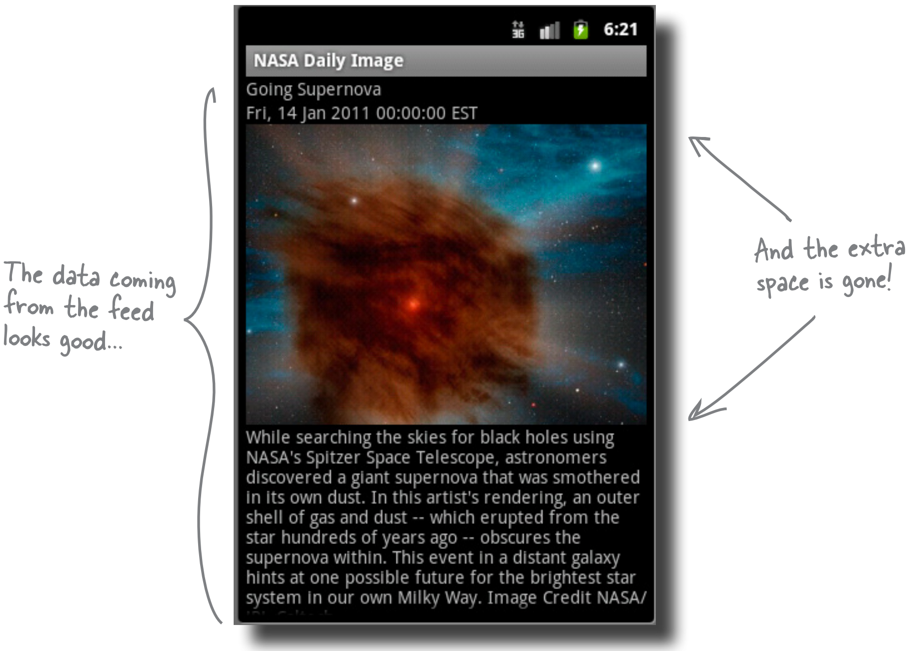
main.xml

```
<ImageView
        android:id="@+id/imageDisplay"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:adjustViewBounds="true" />
```

Set this property in your layout XML.

Test Drive

With the `adjustViewBounds` properts upated in your layout, run the app again. This time, you should see the image resized correctly in the layout.



It's all coming together!

The layout works just like you designed it, the feed parsing is up and running, and the layout issue with the `ImageView` is fixed.



Really great work!

You really did put your new Android development skills to use and built a whole new app! And you learned even more skills along the way. You added scrolling layouts, image resources, and more. But most importantly, you built a *cool* app that **made your users happy**!



Your Android Toolbox

Now that you have a cool RSS feed-parsing app in your toolbox, you can build all kinds of your own cool feed-based apps!

Built-in Problem Solvers

- Use LogCat to view code log statements and errors from your apps.
- Use HierarchyViewer to analyze your views and layouts. This can be extremely helpful when layouts or views aren't behaving as you might expect them to.

View Roundup

- Use TextView to display text. You can use it for small text like labels, or really big text like the Image descriptions.
- Use ImgaeView to display images. You can add your own images to the res directory and display them in an ImageView.
- Use ScrollView to make your content scroll on screen. ScrollView can have only one chid View, so wrap multiple child views in a layout to make them all scroll.

BULLET POINTS

- When working with RSS feeds, download a sample of the feed and decide what content in the feed you want to use in your app.
- Start with `SAX` parsing, but explore the `DOM` and `XMLPULL` parsers to see if they will work better in your app.
- It's a good practice to break your app down into small development pieces. For RSS feed apps that rely on the Internet, it's perfectly acceptable (and even a good idea) to build out your app with test data and plug in the Internet services later.
- Add image resources to the *res/drawable-hdmi* directory (for now). These will get picked up by the Android compiler and the images will be available to your application.
- Use `ImageView` to display images in your app.
- Use `ScrollView` when your app's content is too big for the screen. (Just remember that `ScrollView` can have only one child).
- When things go wrong, use `LogCat` to look at Android errors and log statements.
- Make sure your app has the proper permissions configured in *AndroidManifest.xml*.
- Use `HierarchyViewer` to debug your layouts when your app isn't displaying correctly.

4 long-running processes

When things take time



It would be great if everything happened instantly. Unfortunately, some things just take time. This is especially true on mobile devices, where network latency and the occasionally slow processors in phones can cause things to take a ***bit*** longer. You can make your apps faster with optimizations, but some things just take time. But you ***can*** learn how to **manage long-running processes better**. In this chapter, you'll learn how to show active and passive status to your users. You'll also learn how to perform expensive operations off the UI thread to guarantee your app is always responsive.

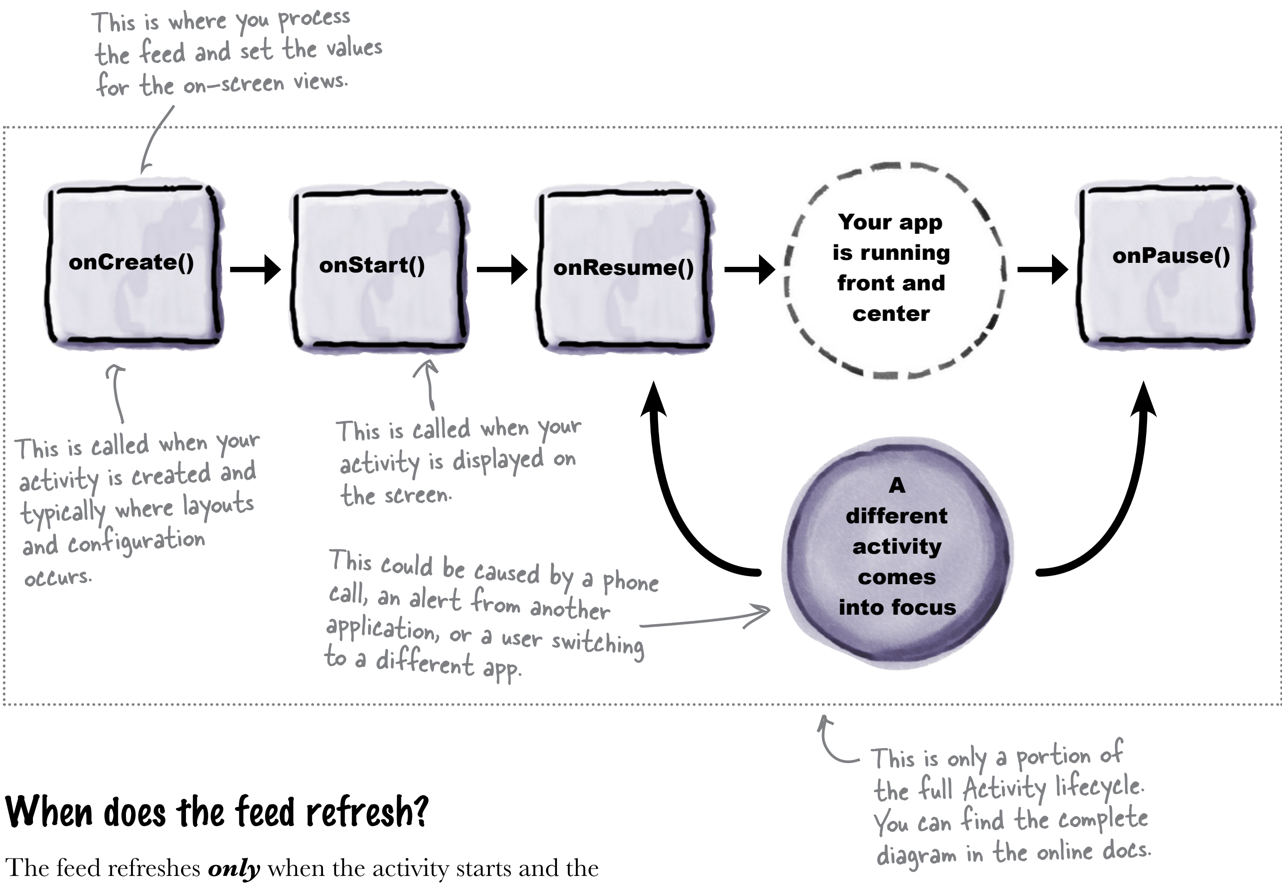


Sounds like a reasonable request...

But why is a refresh button necessary? You'll want to make enhancements to your apps from user feedback, but it's a idea to understand ***why*** you're being asked for something. You have a request to add a refresh button. Let's take a look at the Activity Lifecycle which will explain when the feed loads and why it isn't enough for Bobby...

The Activity Lifecycle

Activity has a number of special methods (called *lifecycle methods*) that get called during the lifecycle of the activity. The **onCreate()** method where you set the layout is one of these methods, and there are ***many*** more. A few of these methods are shown here, so you can see where the feed is (and *is not*) refreshed.



When does the feed refresh?

The feed refreshes ***only*** when the activity starts and the `onCreate()` method is called. The feed will never refresh once the app starts. Currently, the only way to get the app to refresh the feed is to exit the app and then restart it.

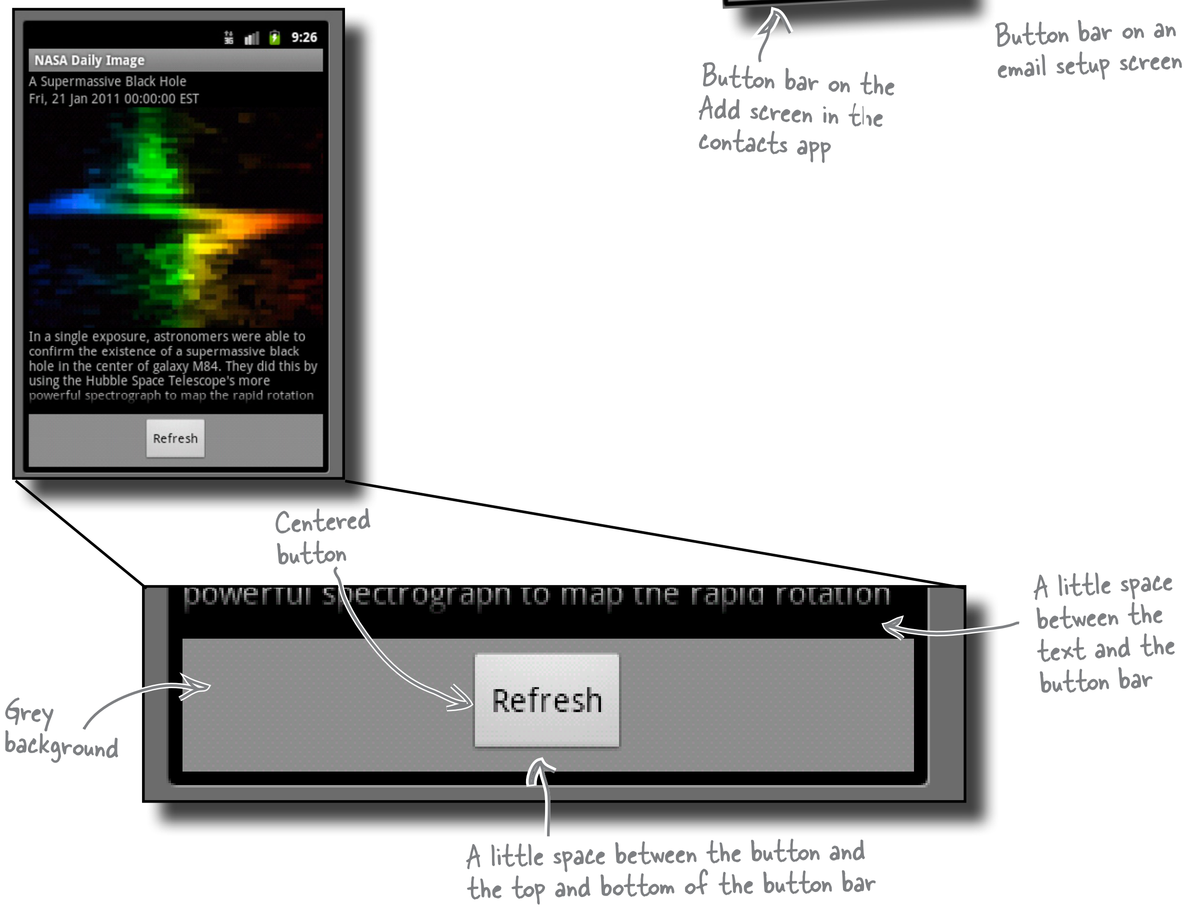
You could override more of the lifecycle methods like `onResume()`, but that would only cover the case where the app is paused and restarted. You could also build some sort of auto-refresh mechanism, but that is very processor and battery intensive. Looks like the refresh button *is* a good idea after all.

Update the user interface

A recurring Android user interface design pattern for on-screen actions, the **button bar** is a gray panel on the bottom of the screen holding one ore more buttons. This will work perfectly for the refresh button placement.

Let's build the **button bar** as a standalone layout and then add it the app's current layout. Encapsulating parts of your fullscreen layout into separate *smaller* layouts can be a good way to organize layouts. And since `LinearLayout` extends `ViewGroup`, which itself extends `View`, you can add your entire new `LinearLayout` you're making for the **button bar** as a child to your original **`ViewGroup`**.





Start with a basic LinearLayout

`LinearLayout` is a surprisingly functional layout manager for basic screen designs. You've already built a few screens using `LinearLayout`, and you're going to build the button bar with it too. You will learn more about `LinearLayout` in the process, and don't worry; you will also learn about other layout managers later in the book.

The key to using `LinearLayout` for the Button Bar is to center the refresh button using the `android:gravity` attribute. Then you can fine-tune the layout.

The beginnings of the button bar layout: right now, just a LinearLayout with a centered button.

```
<LinearLayout
        android:orientation=”horizontal”
        android:layout_width=”fill_parent”
        android:layout_height=”wrap_content”
        android:gravity=”center”
        >
        <Button android:text=”@string/refresh”
               android:layout_width=”wrap_content”
               android:layout_height=”wrap_content” />
</LinearLayout>
```

This is overkill, because horizontal is the default, but it's good to be safe. (annotation on `android:orientation="horizontal"`)



The button is centered.

You're off to a great start! Now start fine-tuning the layout ...

Use properties to fune-tune the layout

With the button properly centered in the layout, let's focus on fine-tuning the layout to get the colors and spacing looking like the button bar examples. Use these properties to get the layout looking the way you want.

padding

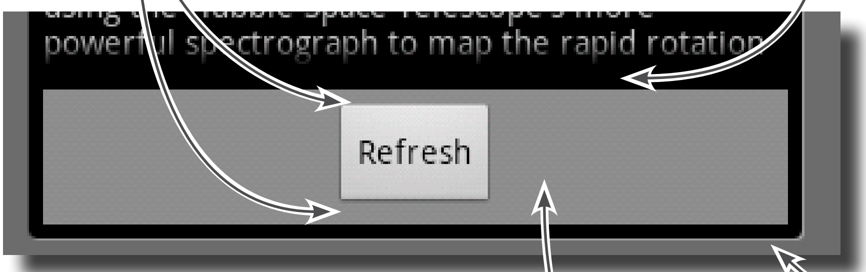
Padding controls the spacing between Views **within** a layout. Use ***Density Independent Pixels*** (DIP) to specify spacing rather than raw pixels to make your layouts ***really*** flexible.

```
android:padding="5dp"
```

margin

Margin controls the spacing between this View and the Views **outside** this layout. Use ***Density Independent Pixels*** (DIP) to specify spacing rather than raw pixels to make your layouts ***really*** flexible.

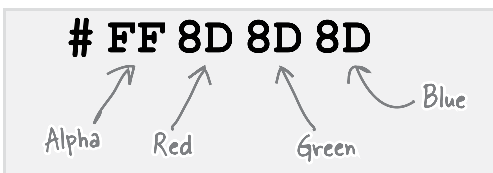
```
android:margin-top="5dp"
```



background

The background property can be set to an image resource, a color, and a few additional Android graphics types. Use a solid color for the button panel background, which is defined in **8-digit hexadecimal** format (two digits each for *alpha*, *red*, *green*, and *blue*).

```
android:background="#ff8D8D8D"
```



layout-width and layout-height

Layout width and height can be set to predefined values of **wrap_content** and **fill_parent**, as well as raw size values in pixels and DIPs. Using **wrap_content** makes the view just as big as it needs to be, while using **fill_parent** sizes the view to fill all of the space it can.

```
android:layout_height="wrap_content"
```

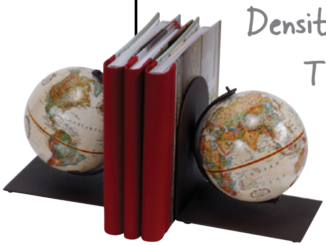
Use wrap_content to size the button. This way, it will be just as big as it needs to fit the "refresh" text.

```
android:layout_height="fill_parent"
```

Use fill_parent for the button bar's LinearLayout width. This will make sure that the layout stretches to the edges of the screen.

the Scholar's Corner

Density Independent Pixels (DIP) Android supports too many screen sizes to keep track! Using raw pixel dimensions in layouts might make your layout look good on one device and terrible on others. Android provides an ABSTRACT sizing measurement called Density Indepent Pixels that is derived from device attributes. This means that you can define layout attributes in DIPs that will look great on all Android devices. Thanks, Android!







Button Bar Layout Magnets

Construct the button bar layout using the magnets below. Think about the width and height for each the button and the `LinearLayout`. And don't worry; you'll have a few extra magnets left over for widths and heights you didn't use.

Here are your magnets.

<Button android:text="@string/refresh"

android:gravity="center"

"wrap-content"

"wrap-content"

"wrap-content"

"fill-parent"

android:background="#ff8D8D8D"

android:layout-height=

android:layout-width=

android:padding="5dp"

android:layout-height=

android:layout-width=

>

<Button android:text="@string/refresh"

/>

android:margin-top="5dp"

<LinearLayout

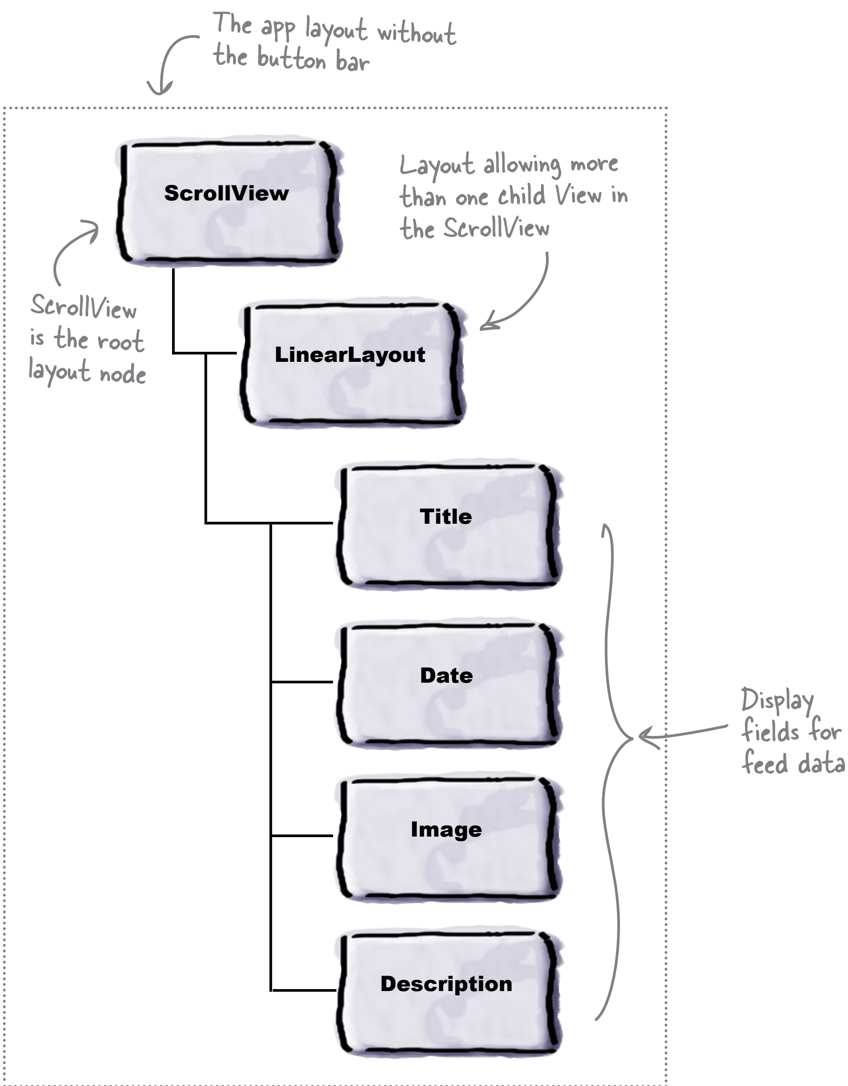
"fill-parent"

</LinearLayout>

"fill-parent"

Exercise

Now that you have the button bar layout, you need to add it to your screen. Below is a graphical representation of your current View/Layout hierarchy. Draw new views and layouts for the button bar Views (and any other views and layouts you need) to complete your layout. Also, remember, just like `ScrollView` that can have only one child, there can be only one root layout.





Button Bar Layout Magnets Solution

You were to construct the button bar layout using the magnets below. Think about the width and height for each the button and the `LinearLayout`. you should have a few extra magnets left over for widths and heights you didn't use.

```
<LinearLayout
    android:layout-width= "fill-parent"
    android:layout-height= "wrap-content"
    android:background="#ff8D8D8D"
    android:margin-top="5dp"
    android:padding="5dp"
    android:gravity="center"
    >
```

The width is set to fill parent, so it fills the width of the screen.

The height is set to wrap content; it shouldn't be the full height (since there is also the scrollpane).

Set the background color to a medium grey.

Add some spacing between the button panel and scroll pane.

Add some spacing around the button inside the layout.

Center the button.

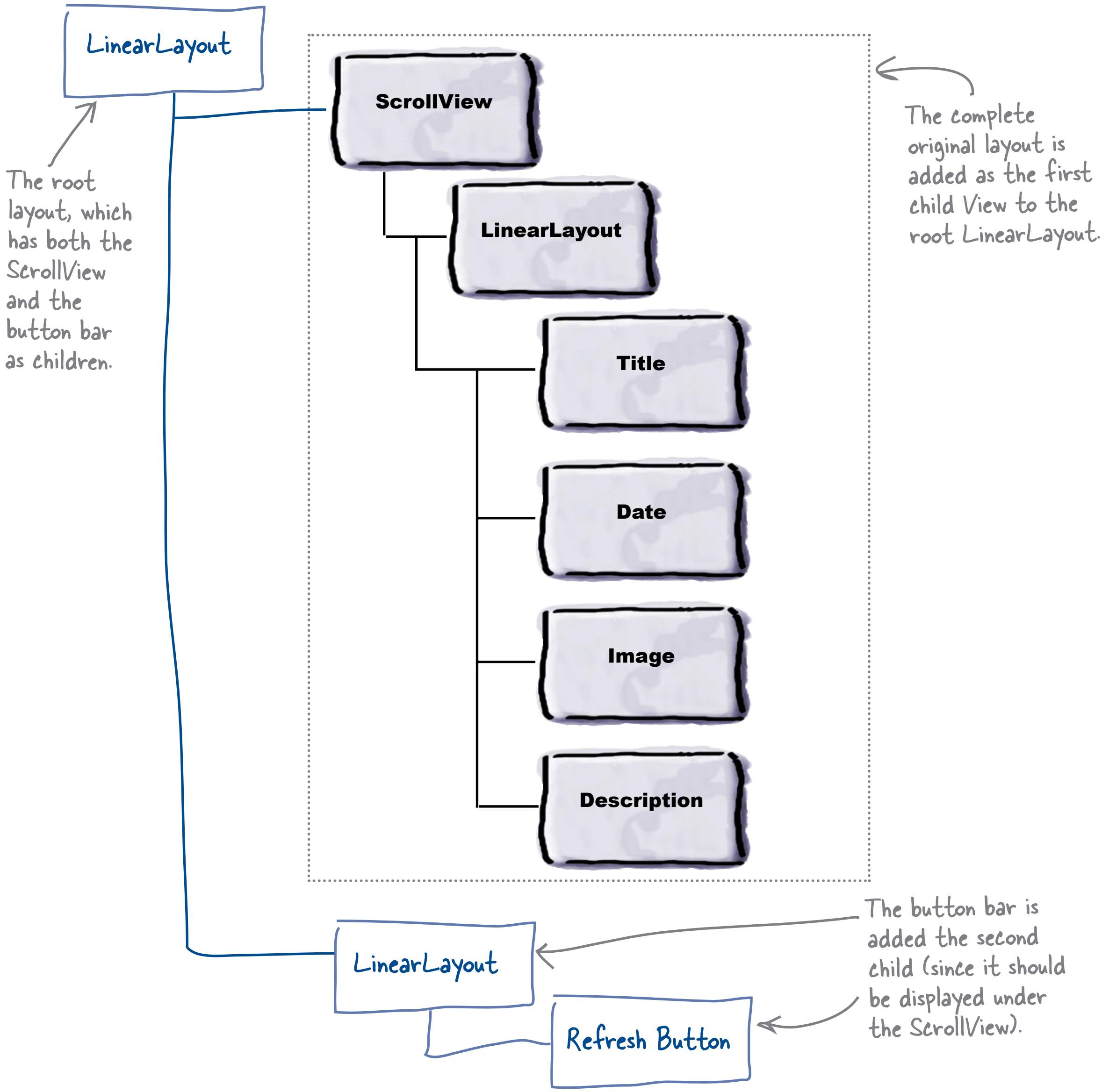
```
    <Button android:text="@string/refresh"
        android:layout-width= "wrap-content"
        android:layout-height= "wrap-content"
    />

</LinearLayout>
```

Both the width and height are set to wrap content, so the button will size as it needs to based on the button text

Exercise Solution

Now that you have the Button Bar layout, you needed to add it to your screen. Below is a graphical representation of your current View/Layout hierarchy. You were to draw new views and layouts for the button bar Views (and any other views and layouts you need) to complete your layout.



Update your app layout

Add the button bar to the app layout in *main.xml*. Also, add the wrapper `LinearLayout` in the root, and add the button bar and the `ScrollView` to that layout.

Do this!

Update your layout in *main.xml*, adding the code for the button bar and the wrapper `LinearLayout`.

Beginning of wrapper layout

```
<LinearLayout  xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:orientation="vertical" >
```

xmlns moved to root layout

Existing layout

```
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" "wrap-content"
        >
<LinearLayout

</LinearLayout>
</ScrollView>
```

Height changed to wrap-content; otherwise, it would fill the screen, leaving no room for the button bar.

(In the ScrollView, `xmlns:android="http://schemas.android.com/apk/res/android"` and `"fill_parent"` in `android:layout_height` are struck through.)

The complete button bar layout

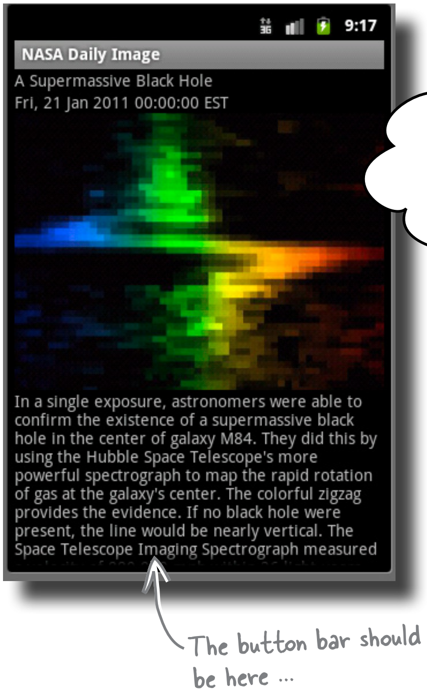
```
<LinearLayout
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:background="#ff8D8D8D"
        android:layout_marginTop="5dp"
        android:padding="5dp" >
        <Button android:text="@string/refresh"
                android:onClick="onRefresh"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content" />
</LinearLayout>
```

```
</LinearLayout>
```

End of wrapper layout

Test Drive

After you update your layout in *main.xml*, run the app to verify your layout updates.





Brain Power

There has ***got*** to be something going on here. The widths and height look OK, and the `LinearLayout` should be resizing everything... right? What could be wrong?

Use LinearLayout's weight property

`LinearLayout` lets you assign a weight property that controls the resizing behavior of its child Views. For the button bar, you want the button bar to be ***just*** as big as it needs to and then have the `ScrollView` **fill** the *entire rest of the screen*.

Weights are defined using the `android:layout_weight` XML attribute and have a number value of **0** or **1**. Using a weight of **1** makes the `View` stretch, while using **0** will make that `View` just as big as needed.

ScrollView definition

```
<ScrollView
      xmlns:android=”http://schemas.android.com/apk/res/android”
      android:layout_width=”fill_parent”
      android:layout_height=”wrap_content”
      android:layout_weight=”1”
```

A weight of 1 fills the screen with just enough space left for the button bar.

Button bar LinearLayout definition

```
<LinearLayout
      android:layout_width=”fill_parent”
      android:layout_height=”wrap_content”
      android:layout_weight=”0”
```

A weight of 0 makes the button bar just as big as needed.

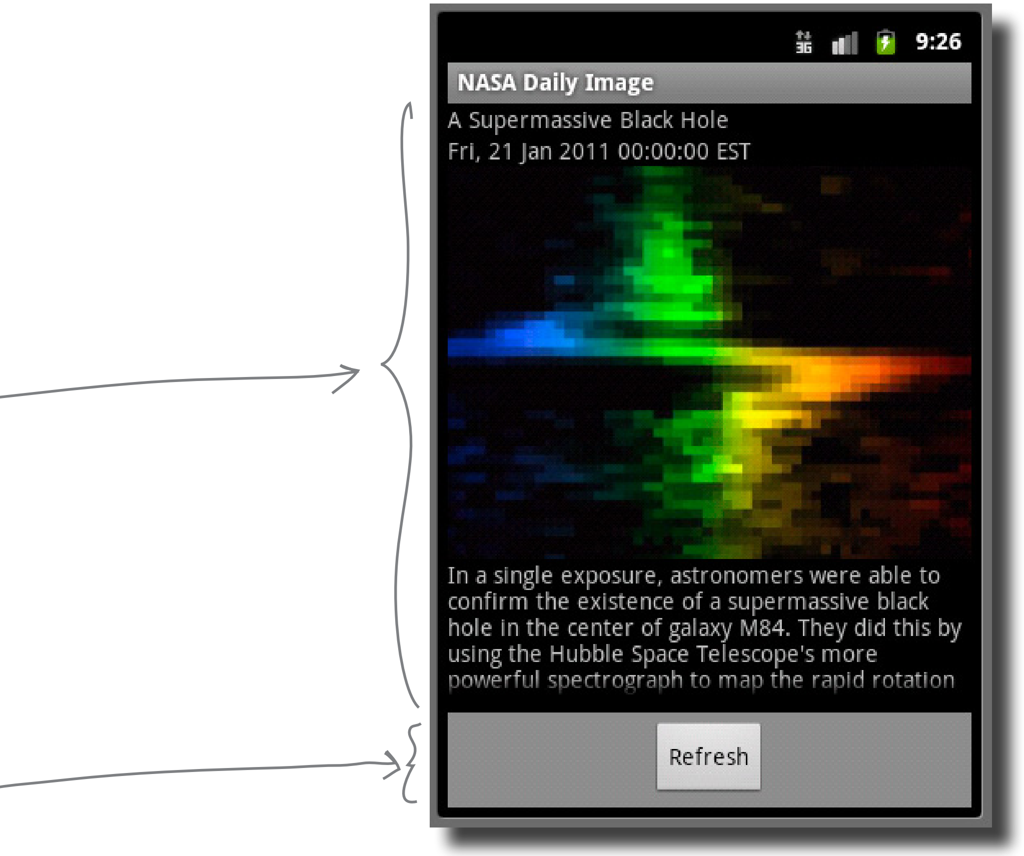
Relax

Where do you find out about these properties?

All of the properties used here (and many, many more) are documented in the Android online documentation. To learn about more of these properties, look at the documentation for your specific layout as well as the layout tutorials. Do a quick search at **developer.android.com**, and you'll get right to it.

Test Drive

Run the app again, and check that the layout weight modifications made the desired layout changed.



Great work!

The app is looking fantastic. Now just wire up the refresh button and you can show it to Bobby.

Connect the refresh button

You already have the feed-handling code working from Chapter 2. To keep your code clean and concise (and without duplicate code), move the feed-handling code to a new method called `refreshFromFeed()`. Then you can call the same feed-processing method from `onRefresh()` ***and*** `onCreate()`.

```
public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        iotdHandler = new IotdHandler();
        iotdHandler.processFeed();
        resetDisplay(iotdHandler.getTitle(),
                iotdHandler.getDate(),
                iotdHandler.getUrl(),
                iotdHandler.getDescription());

        refreshFromFeed();
}
```

Move this code to a new method called refreshFromFeed().

Call refreshFromFeed from onCreate().

```
private void refreshFromFeed() {
      iotdHandler = new IotdHandler();
      iotdHandler.processFeed();
      resetDisplay(iotdHandler.getTitle(),
              iotdHandler.getDate(),
              iotdHandler.getUrl(),
              iotdHandler.getDescription())
}
```

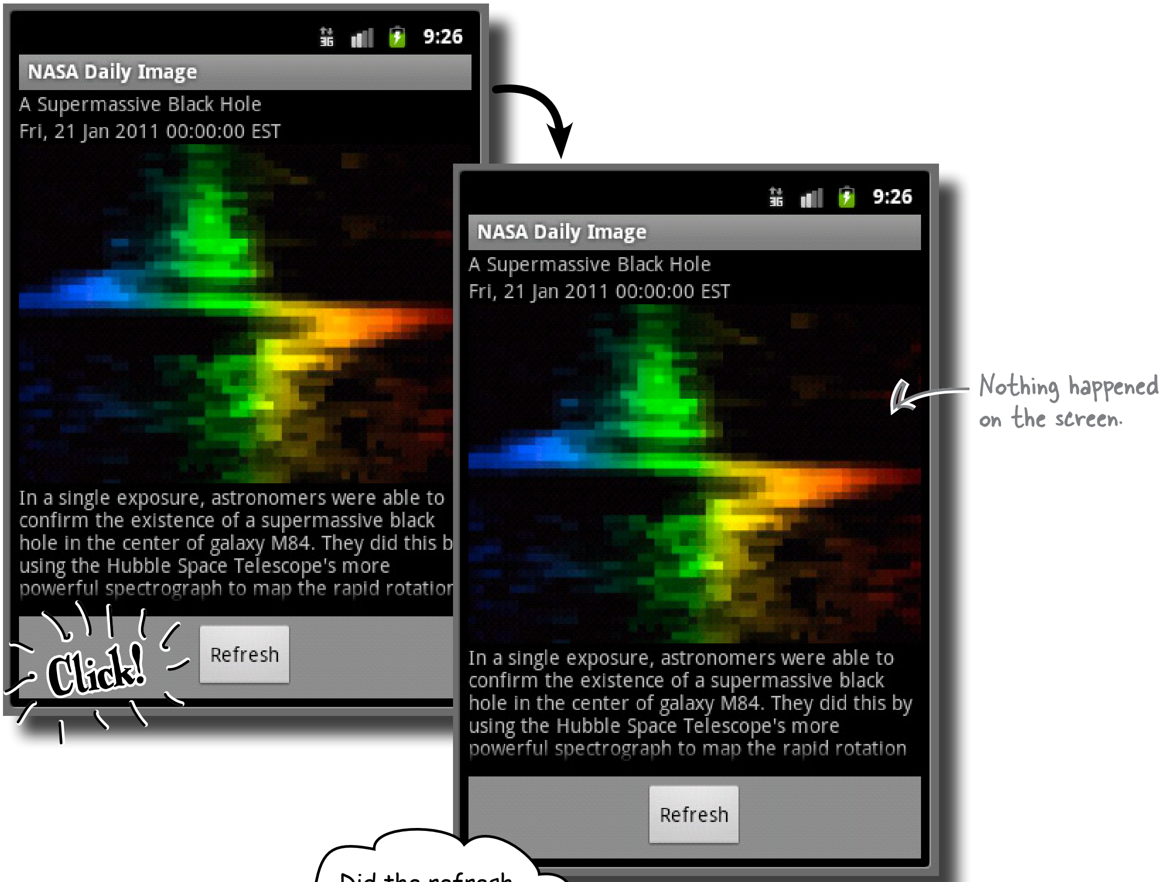
```
public void onRefresh(View view) {
        refreshFromFeed();
}
```

Call the same refreshFromFeed from the button's onRefresh() method.



Test Drive

Run the app again, and click refresh. This will update the app from the feed.





It's not clear what's going on here...

Did the refresh work? Was the feed successfully processed? It's totally unclear what exactly happens here when the user clicks on the refresh button.

Use the debugger

The debugger is an incredibly useful tool for figuring out what's happening while your application is running. The Android Eclipse plugin includes tools to seamlessly use the built-in Eclipse debugger to debug your Android apps, either in the emulator or even on a device. Follow these steps to debug the app and see whether `refreshFromFeed()` is getting called.



This isn't intended to be a detailed debugger tutorial.

There is just enough detail here to debug the NASA app. Take a look at the Android and Eclipse documentation for more tips on using the Eclipse debugger.

1 **Get a breakpoint**

The debugger works by setting stopping points in your app called **breakpoints**. A breakpoint is like a scenic stop on a nice drive where you stop and take a look at what's going on in that spot.

```
            input.close();
            connection.disconnect();
            return bitmap;
        } catch (IOException ioe) {
            return null;
        }
    }

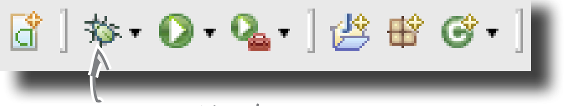
    public void refreshFromFeed() {
        iotdHandler = new IotdHandler();
        iotdHandler.processFeed();
        resetDisplay(iotdHandler.getTitle(), iotdHandler.getDate(), iotdHandler.getU
    }

    public void onRefresh(View view) {
         refreshFromFeed();
```

Double-click in the gray margin to set a breakpoint.

2 **Launch the debugger**

The debug button is just to the left of the play button in the Eclipse toolbar. It uses the Android launch configurations you already set up. Press it to launch the debugger.



Click this button to launch the debugger.

The debugger automatically deploys and attaches to your app (on the emulator or a device).

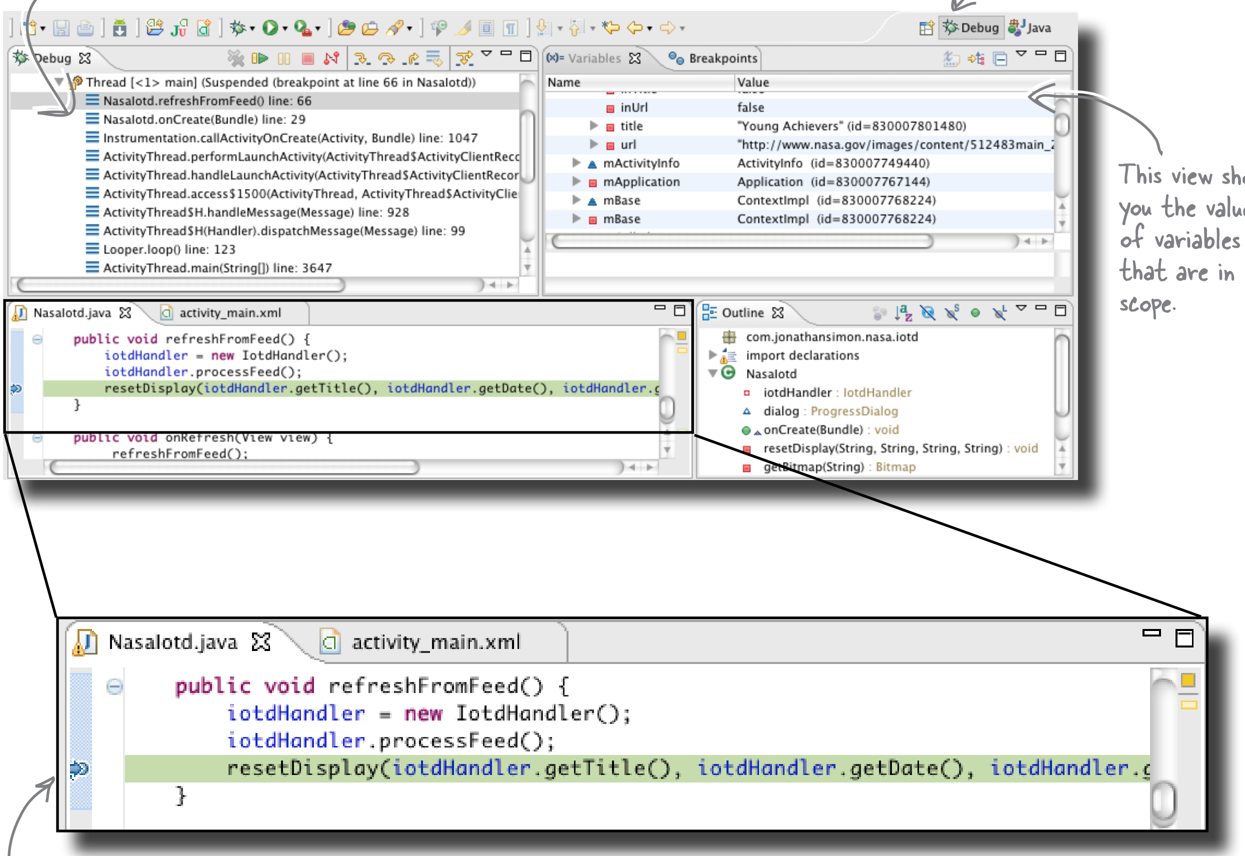
3

Monitor your app in the debug perspective

The debug **perspective** is where you can see the state of your app running. (A **perspective** is Eclipse's name for a stored collection of panels for specific work.) When you launch your app with the debugger, it will immediately hit a breakpoint, because `onCreate()` calls `refreshFromFeed()`, which is where you set your **breakpoint**.

This selector switched from Java to Debug, letting you know you're in the debug perspective. Click Java to take you back to the standard code perspective.

This view shows thread stack traces.



This view shows you the values of variables that are in scope.

The arrow next to the breakpoint indicator lets you know the line was reached.

So the line *was* reached... but how does the *user know*?

Add a progress dialog

The `ProgressDialog` is a utility that shows a **modal** progress pop-up with customized information for your app. `ProgressDialog` is perfect here, because you can show your users status, but you also keep them from repeatedly pressing refresh and successively triggering refresh after refresh.

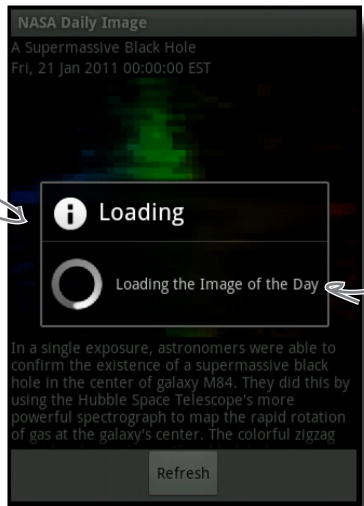
> **Geek Bits**
>
> **Modal** means users can't interact with the application at all. All user input will be ignored.

How do you show a progress dialog?

Show a `ProgressDialog` by calling the static method **`show`** on `ProgressDialog`. The `show` method returns a reference to a `ProgressDialog` isntance. Make sure to cache the reference, as you'll need it to dismiss the dialog when you're done with it.

This is the code to show a progress dialog. Change the title and detail text as needed.

```
ProgressDialog dialog = ProgressDialog.show(
      this,
      "Loading",
      "Loading the image of the Day");
```



Call dismiss on the dialog when you've completed all of your work and the dialog will go away.

```
dialog.dismiss();
```

Call this to dismiss the dialog.

Sharpen your pencil

Below is the `refreshFromFeed` method with long-running code. Add the necessary code to show the `ProgressDialog` before the long-running work is shown. And remember to dismiss the dialog once the work is completed.

```
public void refreshFromFeed() {



        iotdHandler = new IotdHandler();
        iotdHandler.processFeed();
        resetDisplay(iotdHandler.getTitle(),
                iotdHandler.getDate(),
                iotdHandler.getUrl(),
                iotdHandler.getDescription());


}
```

Show the dialog here.

The long-running work of the feed processing.

Dismiss the dialog here, now that all the work is done.

Sharpen your pencil Solution

Below is the refreshFromFeed method with long running code. You were to add the necessary code to show the `ProgressDialog` before the long running work is shown. You should have also dismissed when dialog once the work is completed.

```
public void refreshFromFeed() {

    ProgressDialog dialog = ProgressDialog.show(
        this,
        "Loading",
        "Loading the image of the Day");

    iotdHandler = new IotdHandler();
    iotdHandler.processFeed();
    resetDisplay(iotdHandler.getTitle(),
        iotdHandler.getDate(),
        iotdHandler.getUrl(),
        iotdHandler.getDescription());

    dialog.dismiss();

}
```

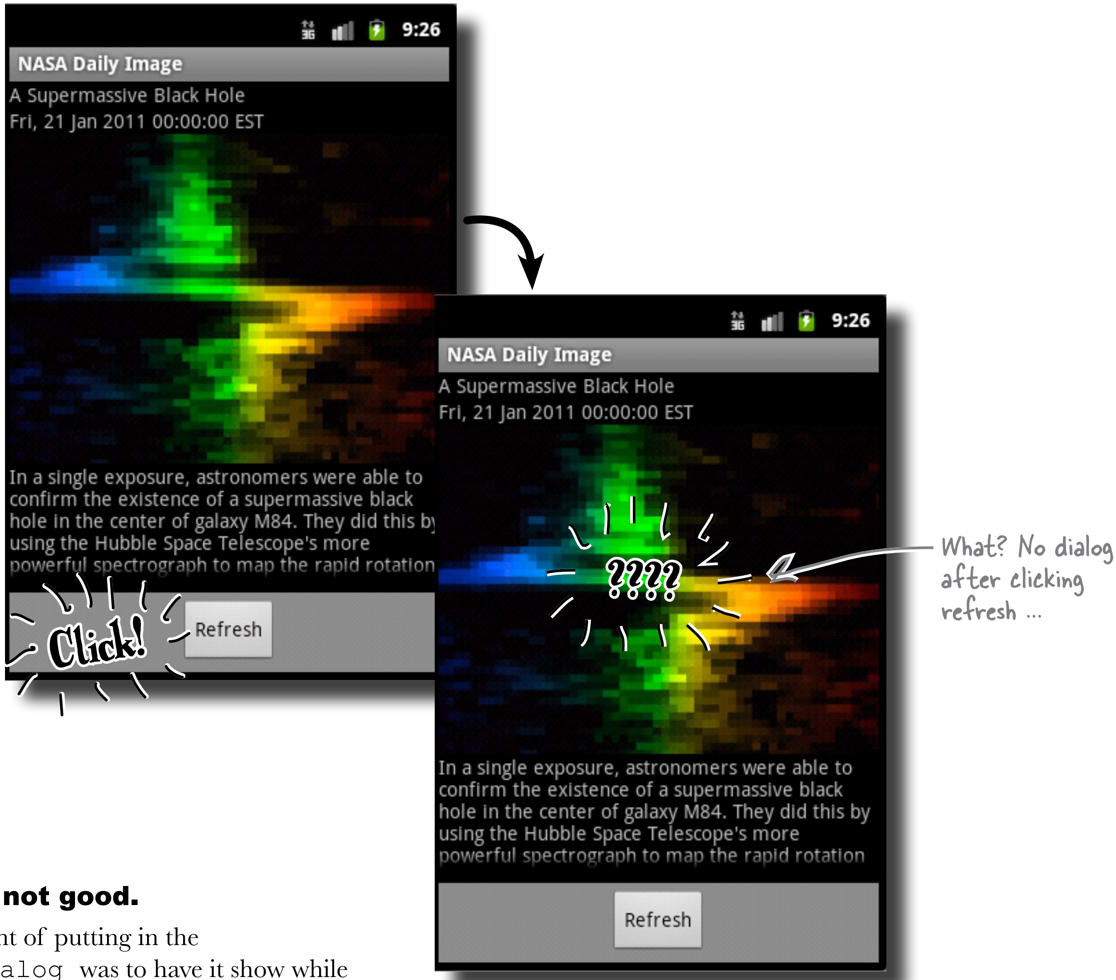
Show the progress dialog.

The feed and UI update code remains untouched.

Dismiss the progress screen, now that the work is done.

Test Drive

Run the app and click Refresh to verify that the `ProgressDialog` is working correctly.



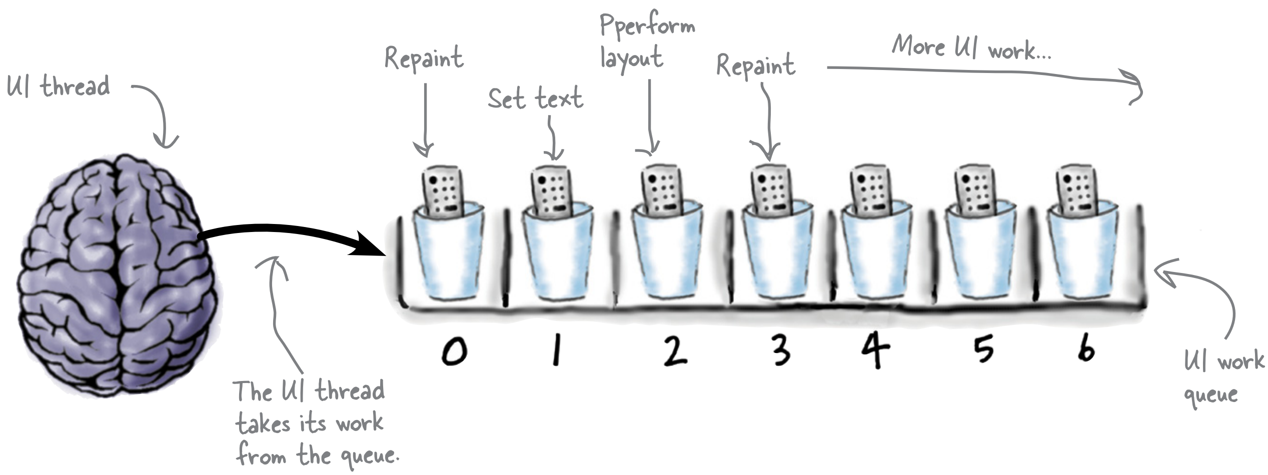
Well that's not good.

The whole point of putting in the `ProgressDialog` was to have it show while the long-running feed-processing work is occurring. The dialog code is in the right place, but for some reason it's not showing. ***What could be happening?***

The problem is in the threading...

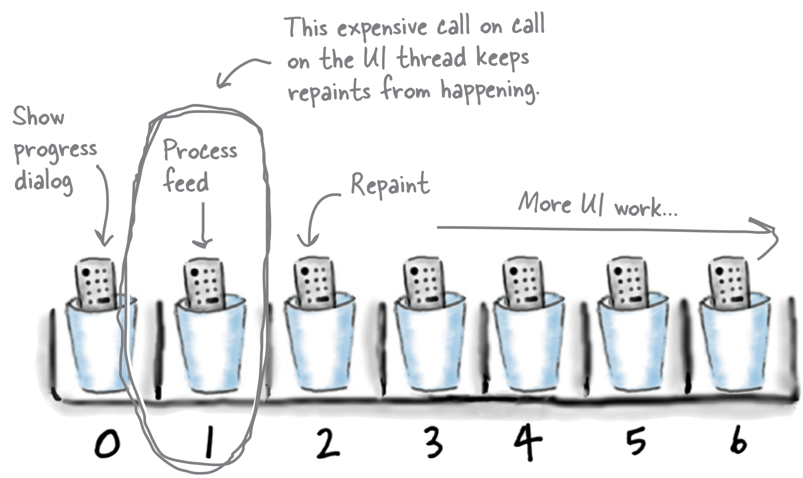
Dedicated UI thread

Android has a dedicated thread for updating the **user interface** (UI). It is responsible for repaints, layouts, and other graphical processing that helps keep the UI responsive and keeps animations smooth. The UI thread has a queue of work, and it continually gets the mot important chunk of work to process.



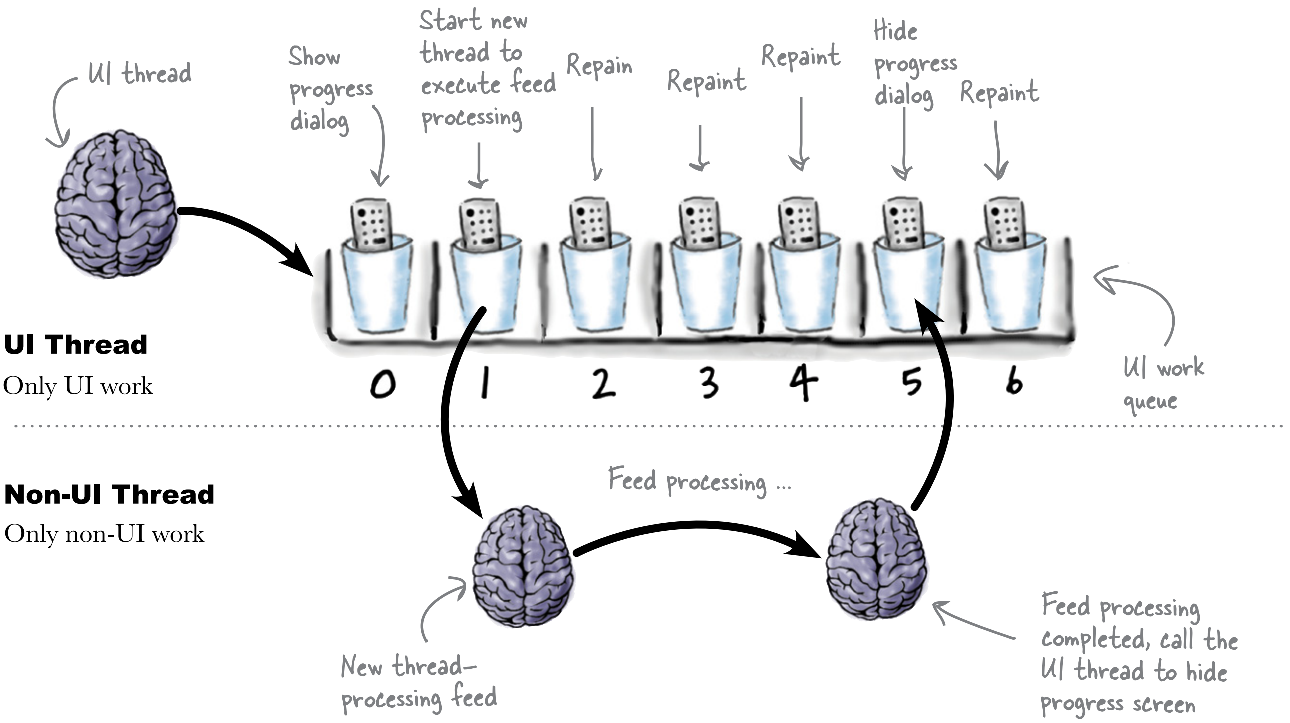
Why didn't the progress dialog display?

The button action occurs in the UI thread by default. When the progress screen is shown, successive calls to repaint the screen are made to support the animation effect. But the process feed code also runs in the UI thread, which occupies the UI thread. By the time the UI thread could run the repaint code, the dialog was hidden.



How do you fix it?

The solution is to keep non-UI work off the UI thread and all UI work on the UI thread.



Moving the feed processing work off the UI thread and onto a separate thread allows the UI thread to focus on repaints. The first repaint shows the progress dialog, and the successive repaints make the animation happen. Then, when the feed processing is completed, the new thread puts an item in the UI queue to hide the progress screen. This switch back to the UI thread is important, because the non-UI thread can't hide the dialog, which is a UI component.

Keep the UI thread free of expensive processing for a responsive UI.

Spawn a new thread for the long process

The most straightforward way to get your long-running processing code on a different thread than the UI thread is to make an inner class extending `Thread` and implementing the `run` method inline.

There are about a million different ways to structure your code to deal with threads. The goal here isn't to debate them, but to understand how to work with the Android UI thread.

```
public void refreshFromFeed() {
        dialog = ProgressDialog.show(
                this,
                "Loading",
                "Loading the Image of the Day");

        Thread th = new Thread() {
                public void run() {
                        if (iotdHandler == null) {
                                iotdHandler = new IotdHandler();
                        }
                        iotdHandler.processFeed();
                        resetDisplay(
                                iotdHandler.getTitle(),
                                iotdHandler.getDate(),
                                iotdHandler.getUrl(),
                                iotdHandler.getDescription());
                        dialog.dismiss();
                }
        };
        th.start();
}
```

Leave this code on the UI thread.

Extend thread.

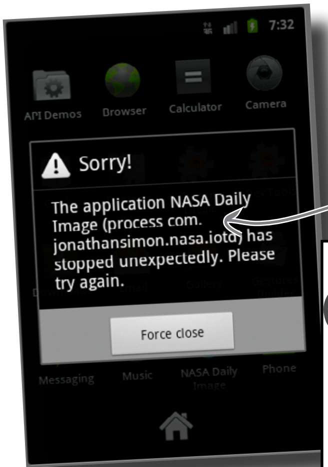
Implement run.

All of the feed-processing goes on the new thread.

Don't forget to start your new thread.

Test Drive

Run the app again, now with the expensive feed-processing code moved to the new thread. The dialog *should* show... but when you run the app, you will see an error.



You'll see this error dialog when you run the app.

The error description

```
FATAL EXCEPTION: Thread-11
android.view.ViewRoot$CalledFromWrongThreadException: Only the original
thread that created a view hierarchy can touch its views.
        at android.view.ViewRoot.checkThread(ViewRoot.java:2932)
        at android.view.ViewRoot.requestLayout(ViewRoot.java:629)
        at android.view.View.requestLayout(View.java:8267)
```

What's the problem?

The problem is the dismissing of the `ProgressDialog`. Properly managing your work on and off the UI thread means ***not only*** getting expensive work **off** the UI thread, but also making sure that all necessary UI code occurs **on** the UI thread.

```
iotdHandler.processFeed();

resetDisplay(
        iotdHandler.getTitle(),
        iotdHandler.getDate(),
        iotdHandler.getUrl(),
        iotdHandler.getDescription());
dialog.dismiss();
```

This needs to occur on the UI thread.

Use Handler to get code on the UI thread

The `dialog.dismiss()` call needs to get back on the UI thread. Getting off of the UI thread is a cinch by creating a new thread. But that thread doesn't have a reference to the UI thread to get code to execute back on the UI thread after the expensive work. That's where **`Handler`** comes in.

`Handler` works by keeping a reference to the thread it was created by. You can pass it work and `Handler` ensures that the code is executed on the instantiated thread. (`Handler` actually works for ***more*** than just the UI thread.)

Start by instantiating a handler from the UI thread

The `onCreate()` method is called from the UI thread. Instantiate the `Handler` there, so you can get work back on the UI thread later.

onCreate method from the Nasalotd Activity

```
Handler handler;

public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        handler = new Handler();
        refreshFromFeed();
}
```

Cache a Handler reference as a member variable, so you don't have to create Handlers over and over again.

onCreate() executes on the UI thread.

Since onCreate() executes in the UI thread, creating the Handler here makes a handler with the ability to execute code on the UI thread.

Pass work to the Handler using post

Once you have a `Handler` instance, you can call `post`, passing it a `Runnable` to execute on the desired thread.

```
handler.post(Runnable runnable)
```

This is a standard Runnable, nothing Android specific.

Get ready to fix refreshFromFeed() with correct threading...



Handler Magnets

Use the magnets below to complete `refreshFromFeed()` with all of the necessary threading changes. The expensive feed-processing code needs to execute on a new thread, and the call to dismiss the dialog has to happen on the UI thread using `Handler`. Assume the `Handler` was already instantiated for you in `onCreate()`.

Here are your magnets.

```
resetDisplay(iotdHandler.getTitle(),
    iotdHandler.getDate(), iotdHandler.getUrl(),
    iotdHandler.getDescription());
```

```
handler.post(
```

```
dialog.dismiss();
```

```
dialog = ProgressDialog.show(this,
    "Loading", "Loading the Image of the Day");
```

```
new Runnable () {
```

```
public void run() {
```

```
iotdHandler.processFeed();
```

```
public void run() {
```

```
}});
```

```
if (iotdHandler == null) {
    iotdHandler = new IotdHandler();
}
```

```
Thread th = new Thread() {
```

```
th.start();
```



Handler Magnet Solution

You were to use the magnets below to complete `refreshFromFeed()` with all of the necessary threading changes. The expensive feed-processing code should be executing on a new thread, and the call to dismiss the dialog should be executing on the UI thread using `Handler`. Assume the `Handler` was already instantiated for you in `onCreate()`.

```
dialog = ProgressDialog.show(this,
        "Loading", "Loading the Image of the Day");
```

The dialog is called from the UI thread (where refreshFromFeed is called from).

Start a new thread for the actual feed code.

```
Thread th = new Thread() {
    public void run() {
        if (iotdHandler == null) {
            iotdHandler = new IotdHandler();
        }
        iotdHandler.processFeed();
```

Post a new Runnable to the Handler.

```
        handler.post(
            new Runnable () {
                public void run() {
```

Call resetDisplay and dismiss the dialog from the UI thread.

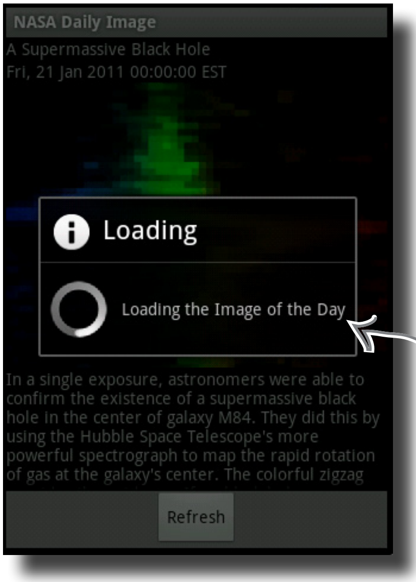
```
                    resetDisplay(iotdHandler.getTitle(),
                        iotdHandler.getDate(), iotdHandler.getUrl(),
                        iotdHandler.getDescription());
                    dialog.dismiss();
        }});
th.start();
```



Test Drive

Now run the app and you'll see the progress screen show while the app loads from the feed during `onCreate()`. You'll also see the the progress screen show when you click the refresh button.

1. Start the app.

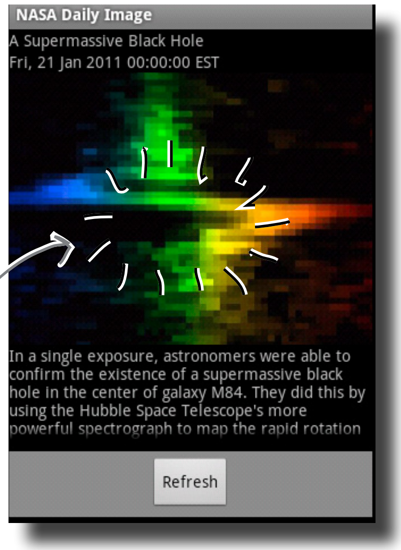


On app startup, the progress dialog will show

2. Give the app a few seconds to load the feed.



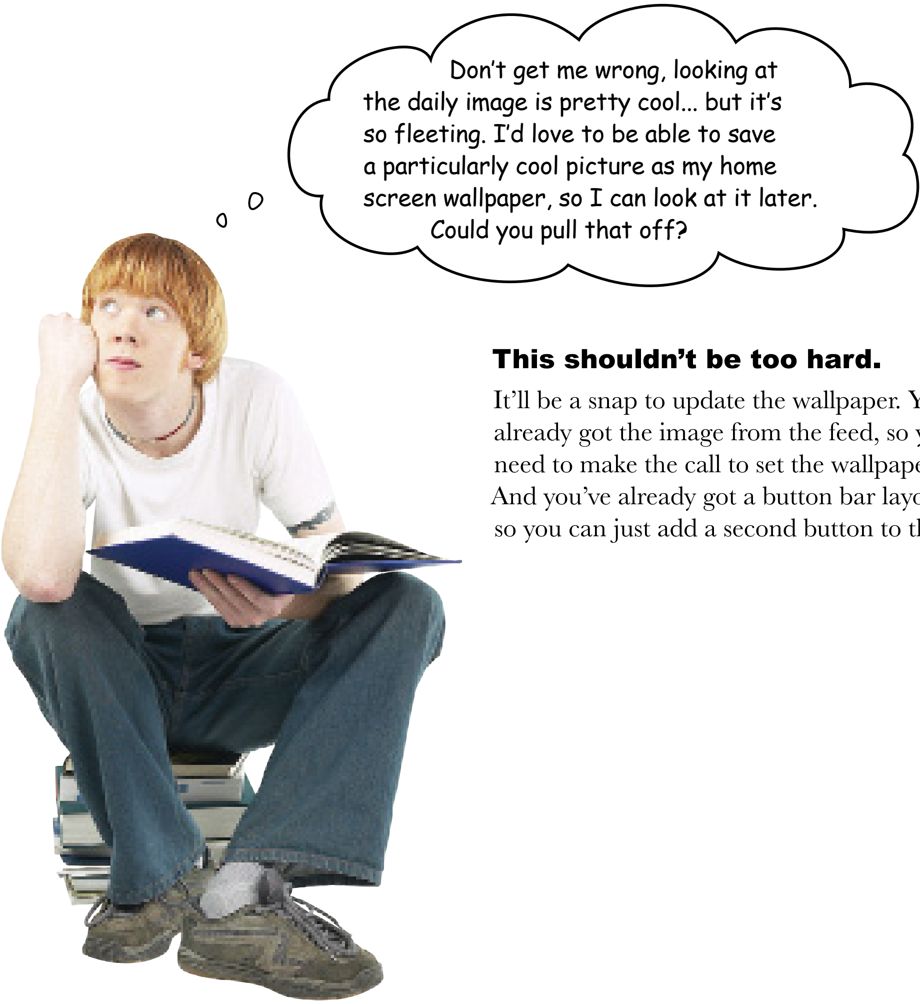
3. Watch the progress dialog get hidden.



No progress dialog, now that the feed processing is complete.

Great work!

Now your users know that the app is doing something. Positive reinforcement goes a long way!



This shouldn't be too hard.

It'll be a snap to update the wallpaper. You've already got the image from the feed, so you just need to make the call to set the wallpaper using that. And you've already got a button bar layout in place, so you can just add a second button to the bar.

The code

You can set the wallpaper by retrieving the `WallpaperManager` and setting the wallpaper by `Bitmap`. You've already got a reference to the `Bitmap` coming from the feed, so this should be a piece of cake.

Ready Bake Code

This is the code to actually set the wallpaper.

```
WallpaperManager wallpaperManager =
      WallpaperManager.getInstance(this);
wallpaperManager.setBitmap( bitmap );
```

You can pass in the bitmap you decoded from the feed here.

"This" refers to the current activity.

The design

You already built the button bar to house the refresh button. And that is an ideal place to add a button to set the wallpaper. (More than two buttons in the button bar could be a problem if the button text is two long, but these two work great.)



The new set wallpaper button

Bobby's going to love this! Let's get started ...

Add the "Set Wallpaper" button

The button bar is built with a `LinearLayout`, so you can just add the new Set Wallpaper button directly to the button bar layout. `LinearLayouts` are horizontal by default, so you can add the **`android:orientation="horizontal"`** or simply rely on the default.

Add the new button to the buton bar layout in *main.xml*:

```
<LinearLayout
      android:orientation="horizontal"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:layout_weight="0"
      android:gravity="center_horizontal"
      android:background="#ff8D8D8D" >

      <Button android:text="@string/refresh"
            android:onClick="onRefresh"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" />

      <Button android:text="@string/setwallpaper"
            android:onClick="onSetWallpaper"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" />
</LinearLayout>
```

LinearLayout defaults to horizontal orientation, but it's a good idea to include the orientation attribute anyway. It makes your code easier to understand later and proptects you in case defaults change.

Add the "Set Wallpaper" button as the second child in the button bar layout. This will add it to the right of the refresh button.

Update *strings.xml* adding the new string for the Set Wallpaper button:

```
<string name="setwallpaper">Set Wallpaper</string>
```

Update the activity for the button action

The feed-processing code already downloads the image from the URL and creates a `Bitmap` from the web resource. To complete `onSetWallpaper` (the `onClick` call declared in the layout), cache the `Bitmap` once decoded and pass that image to the `WallpaperManager`.

```
public class NasaIotd extends Activity {
        private IotdHandler iotdHandler;
        ProgressDialog dialog;
        Handler handler;
        Bitmap image;
```

Make a member variable for the bitmap.

In refreshFromFeed()

```
iotdHandler.processFeed();
image = getBitmap(
        iotdHandler.getUrl());
```

Store the bitmap in the image member variable after processing the feed.

Add the `onSetWallpaper` method to your activity in `NasaIotd.java`:

```
public void onSetWallpaper(View view) {
        Thread th = new Thread() {
                public void run() {
                        WallpaperManager wallpaperManager =
                                WallpaperManager.getInstance(NasaIotd.this);
                        try {
                                wallpaperManager.setBitmap(image);
                        }
                        catch (Exception e) {
                                e.printStackTrace();
                        }
                }
        };
        th.start();
}
```

Setting the wallpaper can take a while, so kick off a new thread to get it off the UI thread.

Since the current scope is an inner class; you can get a reference to "this" by preceding it with the class name.

This will do a default dump of the exception to LogCat.

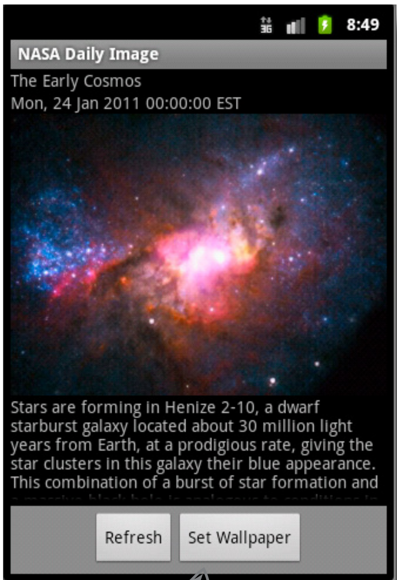
Test Drive

Run the app to make sure the Set Wallpaper button is correctly configured in the layout.

Do this!

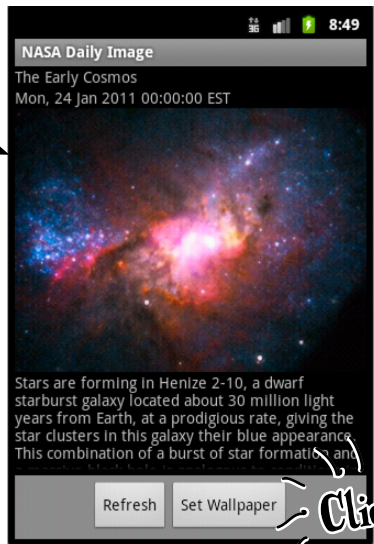
Setting the wallpaper requires a `uses-permission` element with **`android.permission.SET_WALLPAPER`**. Set this now in *AndroidManifest.xml* before you run the app.

First check that the button displays correctly...



The button is displaying correctly, horizontally positioned next to the Refresh button.

The button looks good. Now check and see how it works!



Click!

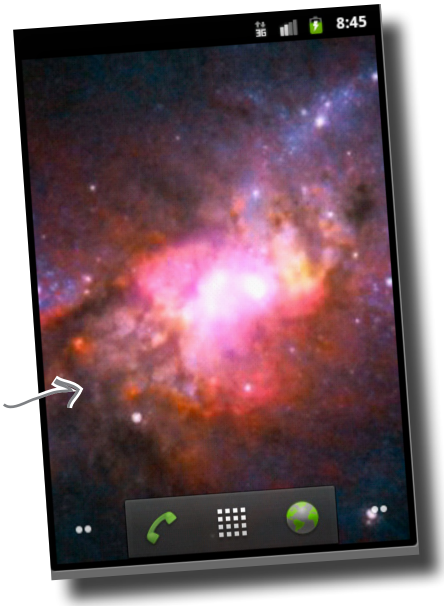


The button did actually work, *but*...

If you go to the home screen, you'll see that the wallpaper was in fact set to the feed image. That said, the **user experience** is *aweful*! Remember that getting your app working is just one part of bigger picture. In order to make successful apps that people want to use (and that will make you bags of money on the Android market!), you need to have a fantastic user experience.

The issue here in setting the the wallpaper is that the change is happening *off screen* away from the user's view. What you need is some **positive reinforcement** so your users ***know*** it worked.

You could just show a ProgressDialog while the wallpaper is being set, but there's a better way...



Use toast to give users reinforcement

You could show a progress screen while the wallpaper is being set. But one of the inherent features of the progress screen is that it blocks users from doing anything. This is great when the feed is loading, because you want to block your users from interacting with the app. (This is what keeps users from repeatedly clicking on refresh.)

But setting the wallpaper is different. You want to make sure to notify your users when the wallpaper is set, but you don't want to keep them from doing something else in the app. For example, it would be perfectly acceptable for the user to set the wallpaper and to scroll down to view the long description while the wallpaper is being set in the background. This wouldn't be possible if you used a progress dialog, because it blocks all user interaction.

Toast

= passive notifications

Progress Dialog

= active, blocking notifications

Android provides Toast for just such occasions

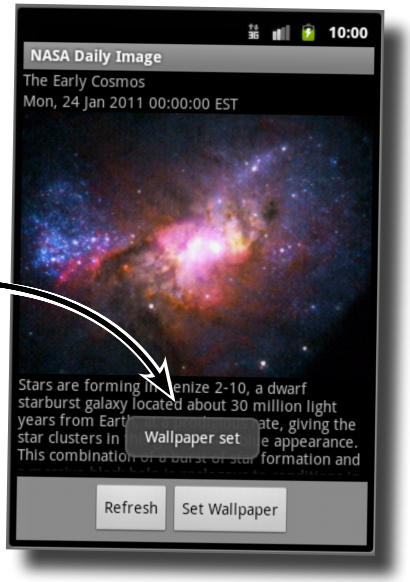
Toast is a passive, non-blocking user notification that shows a simple message at the bottom of the user's screen. A toast typically displays for a few seconds and disappears. Meanwhile, the user can still completely interact with the application. Here is what the app would look like with a Toast message when the wallpaper is set and the code to make it happen.

```
Toast.makeText(this,
        "Wallpaper set",
        Toast.LENGTH_SHORT).show();
```

Pass in your activity. (points to `this`)

Message text (points to `"Wallpaper set"`)

Time to display the toast. (points to `Toast.LENGTH_SHORT`)



Exercise

Complete the `onSetWallpaper` method below adding two `Toast` notifications: one for success and one in case of failure in the `catch` block. The `Toast` call must be made from the UI thread. Use the `Handler` reference cached previously to make both of the toast calls on the UI thread.

```
public void onSetWallpaper(View view) {
       Thread th = new Thread() {
              public void run() {
                     WallpaperManager wallpaperManager =
                            WallpaperManager.getInstance(NasaIotd.this);
                     try {
                            wallpaperManager.setBitmap(image);
```

Add the code here to create the toast confirmation message for setting the wallpaper.

```
                     } catch (Exception e) {
                            e.printStackTrace();
```

Add a toast message in the catch block with the message "Error setting wallpaper."

```
       }}};
       th.start();
}
```

Exercise Solution

You were to complete the `onSetWallpaper` method below adding two `Toast` notifications: one for success and one in case of failure in the `catch` block. The `Toast` call must be made from the UI thread. You should have used the `Handler` reference cached previously to make both of the toast calls on the UI thread.

```
public void onSetWallpaper(View view) {
        Thread th = new Thread() {
                public void run() {
                        WallpaperManager wallpaperManager =
                                WallpaperManager.getInstance(NasaIotd.this);
                        try {
                                wallpaperManager.setBitmap(image);
                                handler.post(
                                        new Runnable () {
                                                public void run() {
                                                        Toast.makeText(NasaIotd.this,
                                                                "Wallpaper set",
                                                                Toast.LENGTH_SHORT).show();
                                }});
                        } catch (Exception e) {
                                e.printStackTrace();
                                handler.post(
                                        new Runnable () {
                                                public void run() {
                                                        Toast.makeText(NasaIotd.this,
                                                                "Error setting wallpaper",
                                                                Toast.LENGTH_SHORT).show();
                                }});
        }}};
        th.start();
}
```

Use the handler to post runnables to the UI thread.

Use NasaIotd.this to get a reference to the Activity from the inner class.

Make a confirmation toast

Show another toast if an exception is caught.

Test Drive

Run the app and click the Set Wallpaper button. Now you will see the wallpaper set *and* a nice toast conformation that lets you and your users know.



Fantastic work! Bobby and all of his friends are going to love this!



The app rocks... you totally came through! I think it's time to share this with more people than just my friends, though. Can you get it on the Market?

Sounds great! Next stop... the Android Market!

You'll have the NASA app up on the market in the next chapter. Stay tuned!

android market



Your Android Toolbox

With proper threading and user feedback, you can guarantee your users a responsive app with a rock solid user experience.

The UI Thread

- Keep expensive work off the UI thread; otherwise, the responsiveness of the UI will suffer.
- Make sure all UI work occurs only on the UI thread. Calling UI code from non-UI threads will throw exceptions throughout your code.

Give your users feedback

- Toast: Use toast to passively display a message to your users
- ProgressDialog: Use a ProgressDialog when you want to block user input and display a message and progress on the screen.

BULLET POINTS

- Use extended properties of `LinearLayout` to fine-tune your screens (`padding`, `margin`, `background`, `gravity`, and more).
- Define layout width and height using `fill_parent` and `wrap_content`. Use `fill_parent` to maximize the size to fill the parent. Use `wrap_content` to make a View just as big as it needs to be.
- Use Density Independent Pixels (DIPs) when you need to define sizing or dimensions. This will ensure your layouts work on the most possible number of devices .
- Layouts can nest (you can add layouts as Views to other layouts). Just remember that too much nesting will slow down the layout and rendering of your screens. So use nested layouts with caution. (You'll learn strategies for this in later chapters.)
- Use the debugger to trace code in the emulator or a device.
- Use a `ProgressDialog` to block users and display progress.
- Use `Toast` to passively notify users of progress.
- Both `Toast` and `ProgressDialog` can be extensively customized for your app.
- Keep expensive work off the UI thread, and UI work only on the UI thread
- Use `Handler` to add UI work to the UI thread's queue from non-UI threads.

5 multiple device support

Run your app everywhere

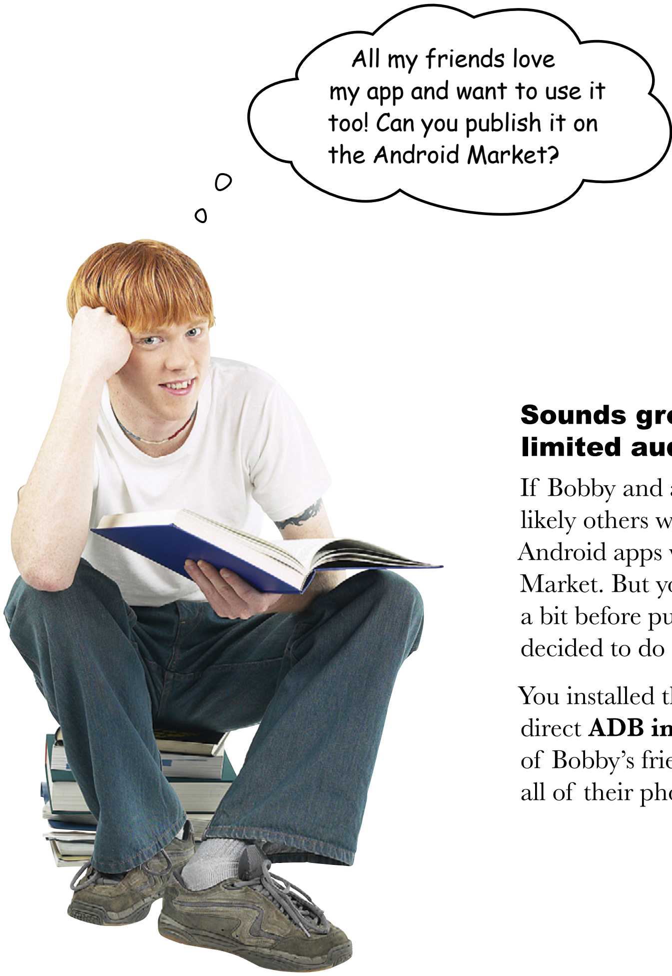


There are a lot of different sized Android devices out there.

You've got big screens, little screens, and everything in between. And it's your job to support them all! Sounds crazy, right? Right now you're probably thinking "***How can I possibly support all of these different devices***?" But with the right strategies, you'll be able to target all of these devices ***in no time*** and with **confidence**. In this chapter, you'll learn how Android classifies different devices into groups based on **screen size** as well as **screen density**. Using these groups, you'll be able to make your app look great on these different devices, and best of all, with a **manageable** amount of work!

Bobby and all of this friends love the app!

Bobby has been using the NASA image app all around schools and his friends have all been asking him for a copy.



Sounds great... but how about a limited audience?

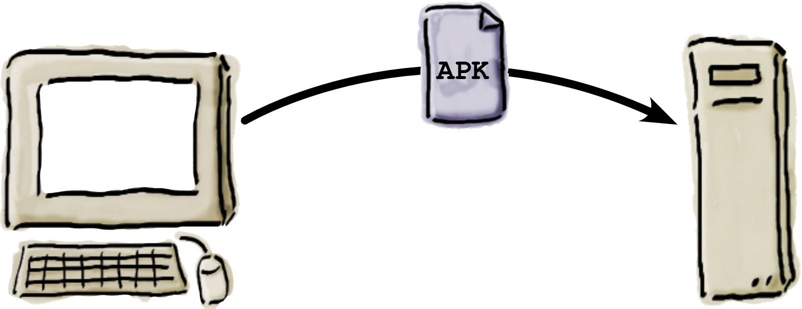
If Bobby and all of this friends want the app, likely others would too. And the place to share Android apps with everyone is the Android Market. But you would like to test the app out a bit before publishing it for the world. So you decided to do

You installed the app on Bobby's phone using the direct **ADB install**, but you can't do that with all of Bobby's friends since you don't have access to all of their phones.

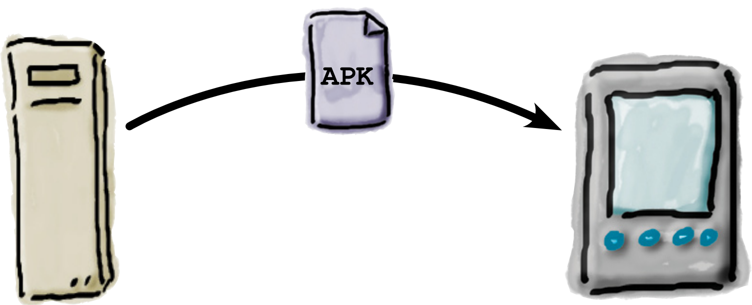
Can you share the app without using the market?

Sure! You can publish the apk on any webserver. Then anyone can download the app by navigating to the hosted APK on their Android device.

1. **Upload your APK to a webserver**
You can upload the APK to any webserver. You can find the APK in your project's bin directory and transfer it to your webserver. (Note: You'll need to add the mime type `application/vnd.android.package-archive` for the `.apk` extension or have your web administrator do this for you).



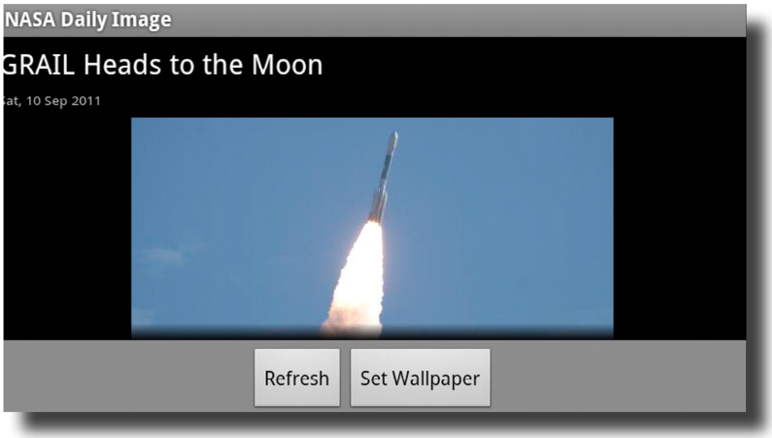
2. **Navigate to the URL on the device**
Anyone who wants to install the app can navigate to the URL of the hosted APK from the browser on their device. This will download and install the app for them. (Note: Each user will have to configure the 'Unknown sources; setting to allow non-market applications on their device).



Let's get some of Bobby's friends to download the app...

Let's see what Bobby's friends have to say

Bobby got a bunch of his friends to download the app over the air and play with it for a few days. Most people were pretty happy. But two of his friends, Jesse and Shawn, came back with some great suggestions for improvement.



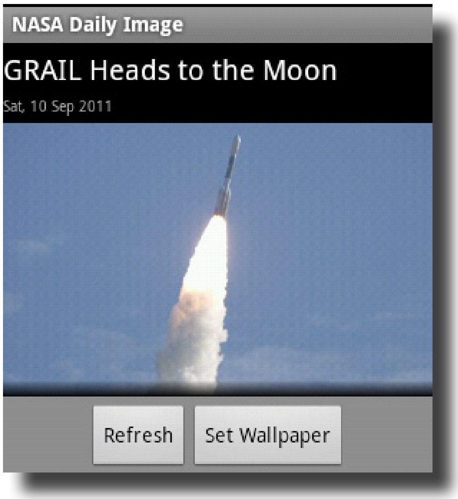


Jesse's wants to see more of the image in landscape mode

Jesse has a phone with slide-out keyboard, which forces the app into landscape mode. Technically it works, but Jesse doesn't like how much vertical space the buttons are taking up. She would love to see more of the images instead of those buttons...

Shawn wants small screen phones to show more of the image too

Shawn has a *really* small phone (300x350 pixels to be exact). Like Jesse, shawn thinks the buttons on the bottom are a waste of space on his ***extremely*** small phone. He's love to see those buttons moved somewhere too.





Shawn also pointed out that the home icon is pretty boring...

Android uses a default icon on the home screen. It's pretty boring though. Shawn really thinks you should update it to make the app look more polished.



The home icon is boring.

So many devices, and so many issues!

You knew there are all kinds Android devices out in the wild with different sizes and resolutions. But with such a simple layout, who would have though there would be so many issues?

Some of the issues are also device specific

Jesse and Shawn both have suggestions for improving the app in landscape mode and for really small screened devices. But you don't want to change the regular app in portrait mode. The app you built at the end of Chapter 4 still works great for those devices.



On Android, you *can* make changes just for specific devices!

With all of the different device shapes and sizes in the vast world of Android devices, you'll often need to customize your apps for a few devices, like really big, or really small screens. Luckily, Android provides a mechanism for using a default layout and overriding those layouts for specific devices.

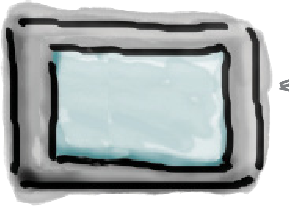
Make a plan

Making your app work on all kinds of Android devices takes some careful planning. In the case of the NASA Image of the Day app, Bobby's friends tested it out on all kinds of different devices and you've narrowed down just a few cases where you need to improve.

Here's what you're going to do to get this app market ready in no time!

Update the layout for landscape mode

You can solve Jesse's problem by creating a special layout for landscape mode. This way, you can leave the regular portrait screen as it was and make adjustments for the landscape version.



Update the layout for small screens

Shawn brought up a good point that the buttons are wasting space on small screens. But just like landscape mode, you want to be able to leave the regular layout alone and just make the modification for small screens.



Update the icon

Shawn also pointed out the boring default Android home icon. Since you're goal is to get the app Android-market-ready, let's get that fixed while you're at it.



Turn the page to get started! →

Preview landscape mode in your emulator

The first issue to address is the lack of vertical space in landscape mode. But before you can fix anything, you need to be able to duplicate the issue reported by your users in your own development environment. In this case, you need to be able to view the app in landscape mode.

You can do this in any running Android emulator by pressing `CTRL` → `F12`.

Do this!

Press `CTRL` → `F12` to switch your emulator between portrait and landscape modes.

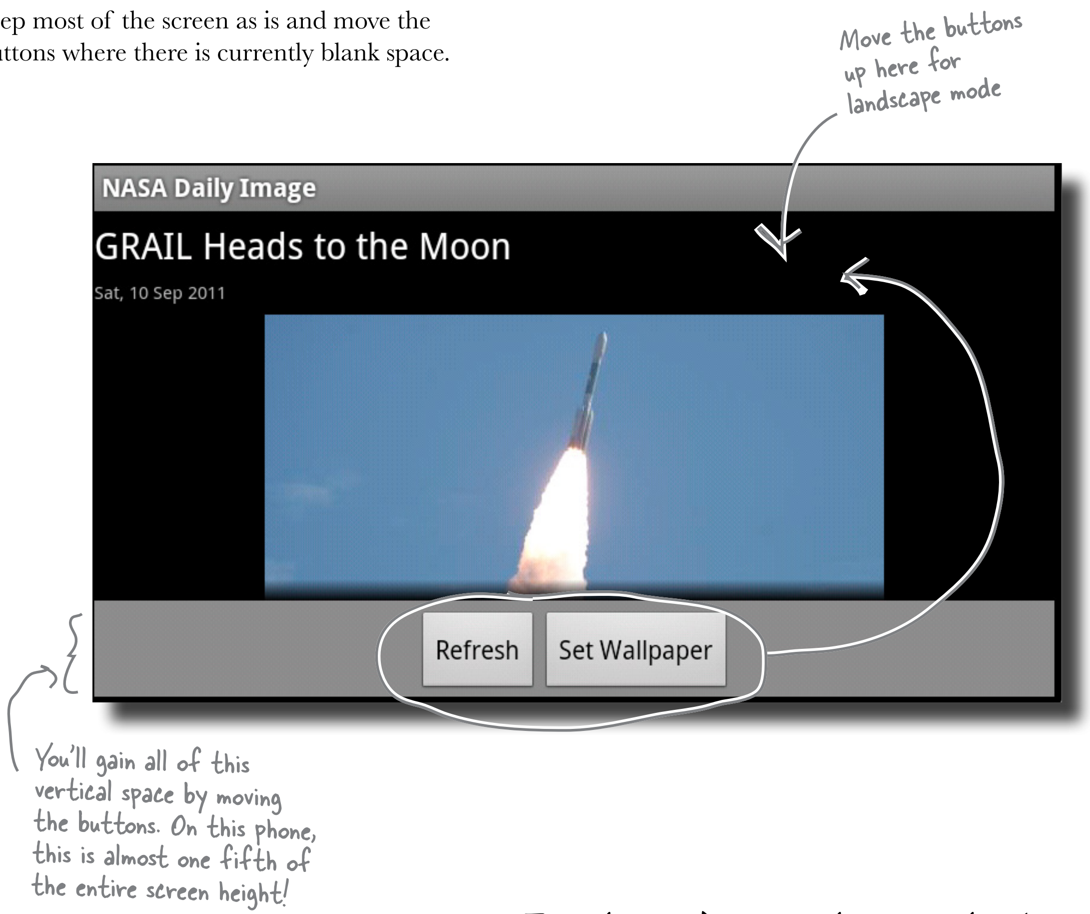


Update the design for landscape mode

The main issue with landscape mode is the buttons. With the button bar gone, you'll gain a lot more vertical space to show the day's image.

But where could you put those buttons?

There are a number of different solutions, but let's move the buttons to the top right of the screen in line with the title and date. This will keep most of the screen as is and move the buttons where there is currently blank space.

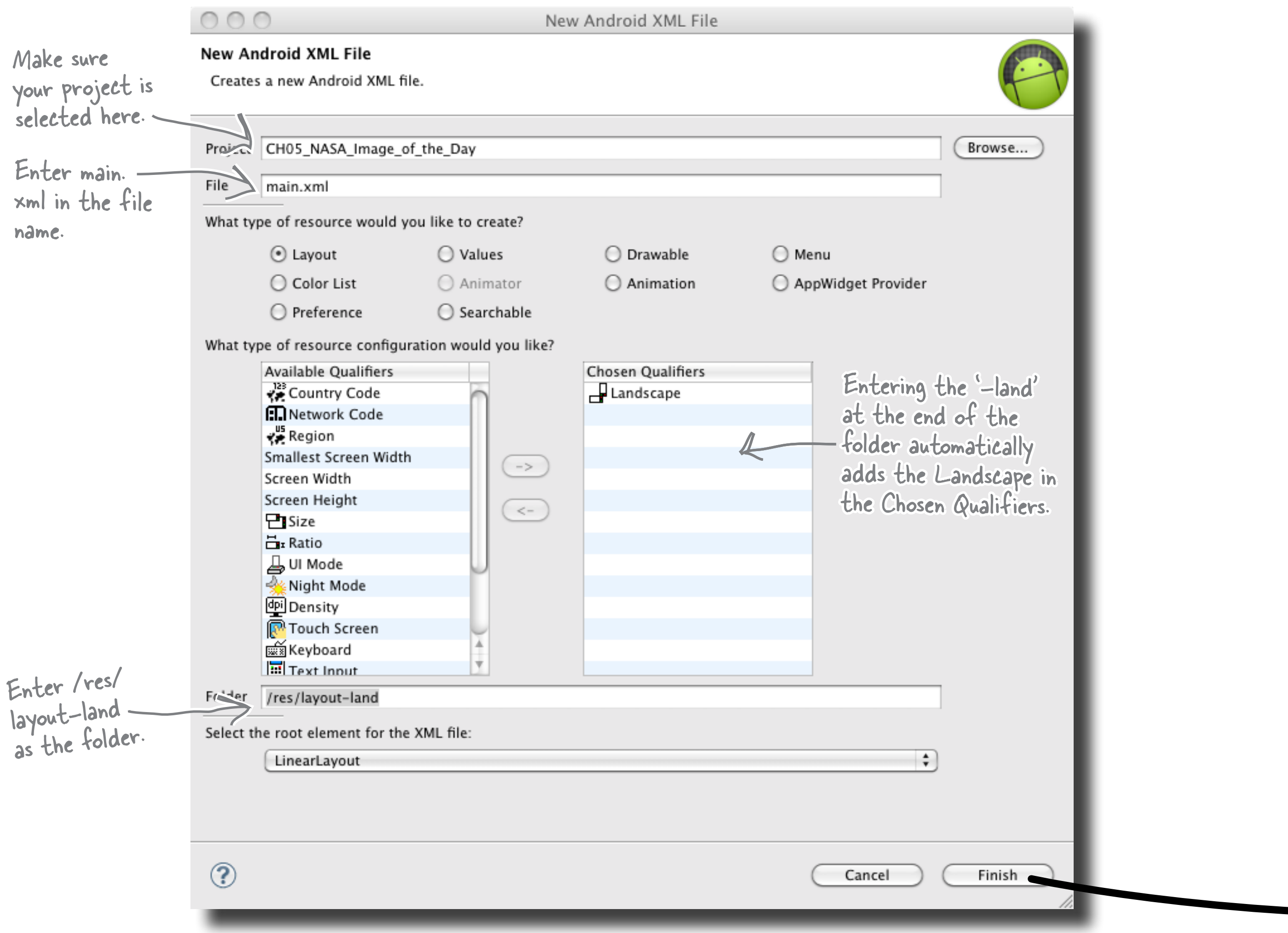


But how do you change the layout *just* for landscape mode?

Create a landscape layout with the wizard

You can created new layouts using the New Android XML File wizard. This wizard isn't specific to layouts, you can use it to make all kinds of different Android XML resource files. Launch the wizard by going to File → New → Android XML File.

The project will be filled in for you. Select the "Layout" radio button as the resource type and enter "main.xml" as the file name. Then for the folder enter "/res/layout-land". This will automatically add "Landscape" as a Chosen Qualifier.



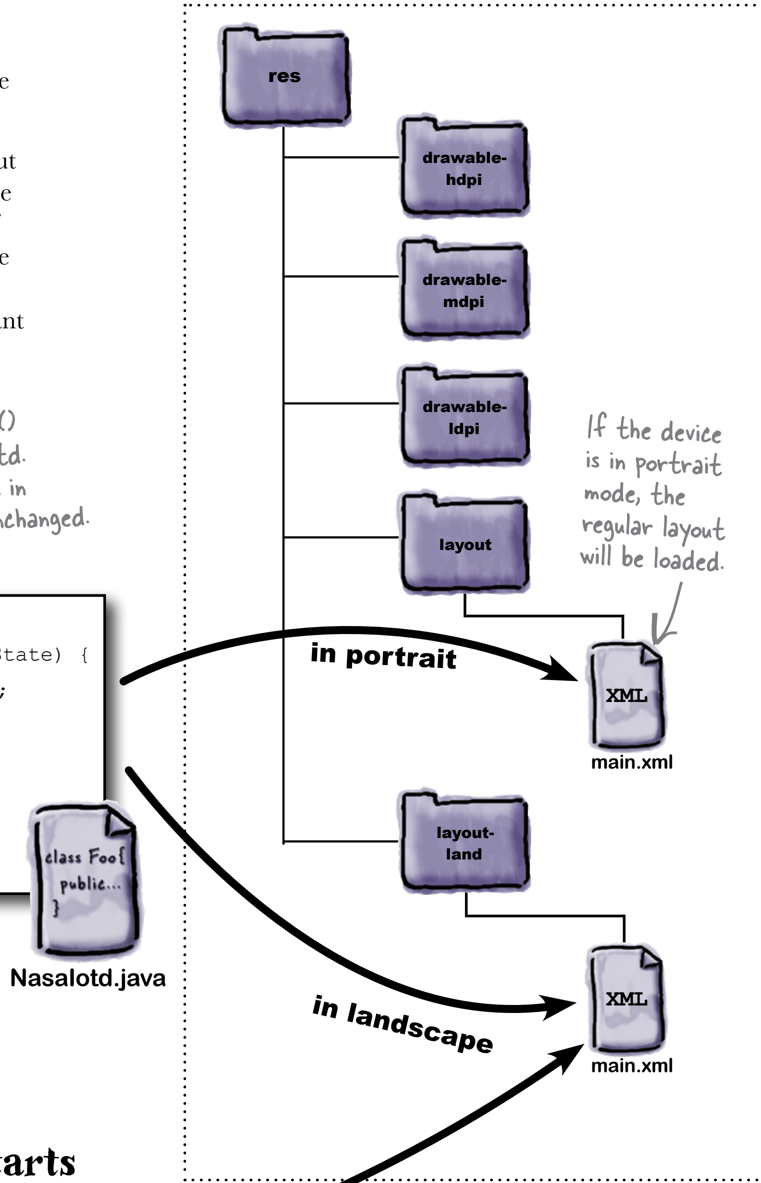
Where is the layout?

The new layout you made is named the same as your existing layout, but your new layout is in a parallel directory called layout-land. This special construct allows the Android runtime to determine the best layout based on the device's state.

If the device is in proptrait mode, it loads the layout at `/res/layout/main.xml`. And if the device is in landscape mode, it loads the layout at `/res/layout-land/main.xml`. This doesn't require any code changes to your Activity since both resources are still referenced by the same R constant at `R.layout.main`.

Here is the onCreate() method from Nasalotd.java. The R constant in setContentView is unchanged.

```
public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        handler = new Handler();
        refreshFromFeed();
}
```



The layout you just created starts out empty... time to build it out!



Landscape Layout Magnets

Since the portrait and landscape layouts are so similar, a good starting place is to copy and paste the layout. But some things will have to change too. Below is the copied beginning and end of the layout. Use the magnets below to complete the layout with the buttons on the top right of the screen.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" >
```

This is the root LiearLayout.

Here are some magnets.

```
</LinearLayout>
```

```
</LinearLayout>
```

```
<Button android:text="@string/refresh"
        android:onClick="onRefreshButtonClicked"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/refreshButton" />
```

```
<LinearLayout
        android:orientation="vertical"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:gravity="left"
        android:layout_weight="1" >
```

Here are some MORE magnets.

```
</LinearLayout>
```

```
<LinearLayout
        android:orientation="horizontal"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:gravity="left" >
```

```
<TextView
        android:id="@+id/imageDate"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:textSize="10dp"
        android:layout_marginBottom="5dp" />
```

```
<Button android:text="@string/setwallpaper"
        android:onClick="onSetWallpaper"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/setWallpaperButton" >
```

```
<LinearLayout
        android:orientation="horizontal"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:gravity="center_vertical"
        android:layout_weight="0"
        android:layout_marginTop="5dp" >
```

```
<TextView
        android:id="@+id/imageTitle"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:textSize="20dp"
        android:textColor="@color/image_title_color"
        android:layout_marginTop="5dp" />
```

Here are EVEN MORE magnets.

The ScrollView and its contents remain unchanged.

```
<ScrollView android:layout_width="fill_parent"
        android:layout_height="wrap_content"android:layout_weight="1" >
        <LinearLayout android:orientation="vertical"
                android:layout_width="fill_parent"
                android:layout_height="wrap_content"
                android:gravity="center_horizontal" >
                <ImageView android:id="@+id/imageDisplay"
                        android:layout_width="wrap_content"
                        android:layout_height="wrap_content"
                        android:layout_marginBottom="5dp"
                        android:adjustViewBounds="true" />
                <TextView android:id="@+id/imageDescription"
                        android:layout_width="wrap_content"
                        android:layout_height="wrap_content" />
        </LinearLayout>
</ScrollView>

</LinearLayout>
```



Landscape Layout Magnet Solution

Since the portrait and landscape layouts are so similar, a good starting place is to copy and paste the layout. But some things will have to change too. Below is the copied beginning and end of the layout. You should have used the magnets below to complete the layout with the buttons on the top right of the screen.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" >
```

```
<LinearLayout
        android:orientation=”horizontal”
        android:layout_width=”fill_parent”
        android:layout_height=”wrap_content”
        android:gravity=”left” >
```

This is a horizontal layout for the entire header.

```
<LinearLayout
        android:orientation="vertical"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:gravity="left"
        android:layout_weight="1" >
```

Here is a vertical LinearLayout for the title and date.

```
<TextView
        android:id="@+id/imageTitle"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:textSize="20dp"
        android:textColor="@color/image_title_color"
        android:layout_marginTop="5dp" />
```

Add the title and date view.

```
<TextView
        android:id="@+id/imageDate"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:textSize="10dp"
        android:layout_marginBottom="5dp" />
```

```
</LinearLayout>
```

```
<LinearLayout
        android:orientation="horizontal"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:gravity="center_vertical"
        android:layout_weight="0"
        android:layout_marginTop="5dp" >
    <Button android:text="@string/refresh"
            android:onClick="onRefreshButtonClicked"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:id="@+id/refreshButton" />
    <Button android:text="@string/setwallpaper"
            android:onClick="onSetWallpaper"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:id="@+id/setWallpaperButton" >
</LinearLayout>
<ScrollView android:layout_width="fill_parent"
        android:layout_height="wrap_content"android:layout_weight="1" >
        <LinearLayout android:orientation="vertical"
                android:layout_width="fill_parent"
                android:layout_height="wrap_content"
                android:gravity="center_horizontal" >
                <ImageView android:id="@+id/imageDisplay"
                        android:layout_width="wrap_content"
                        android:layout_height="wrap_content"
                        android:layout_marginBottom="5dp"
                        android:adjustViewBounds="true" />
                <TextView android:id="@+id/imageDescription"
                        android:layout_width="wrap_content"
                        android:layout_height="wrap_content" />
        </LinearLayout>
</ScrollView>
</LinearLayout>
</LinearLayout>
```

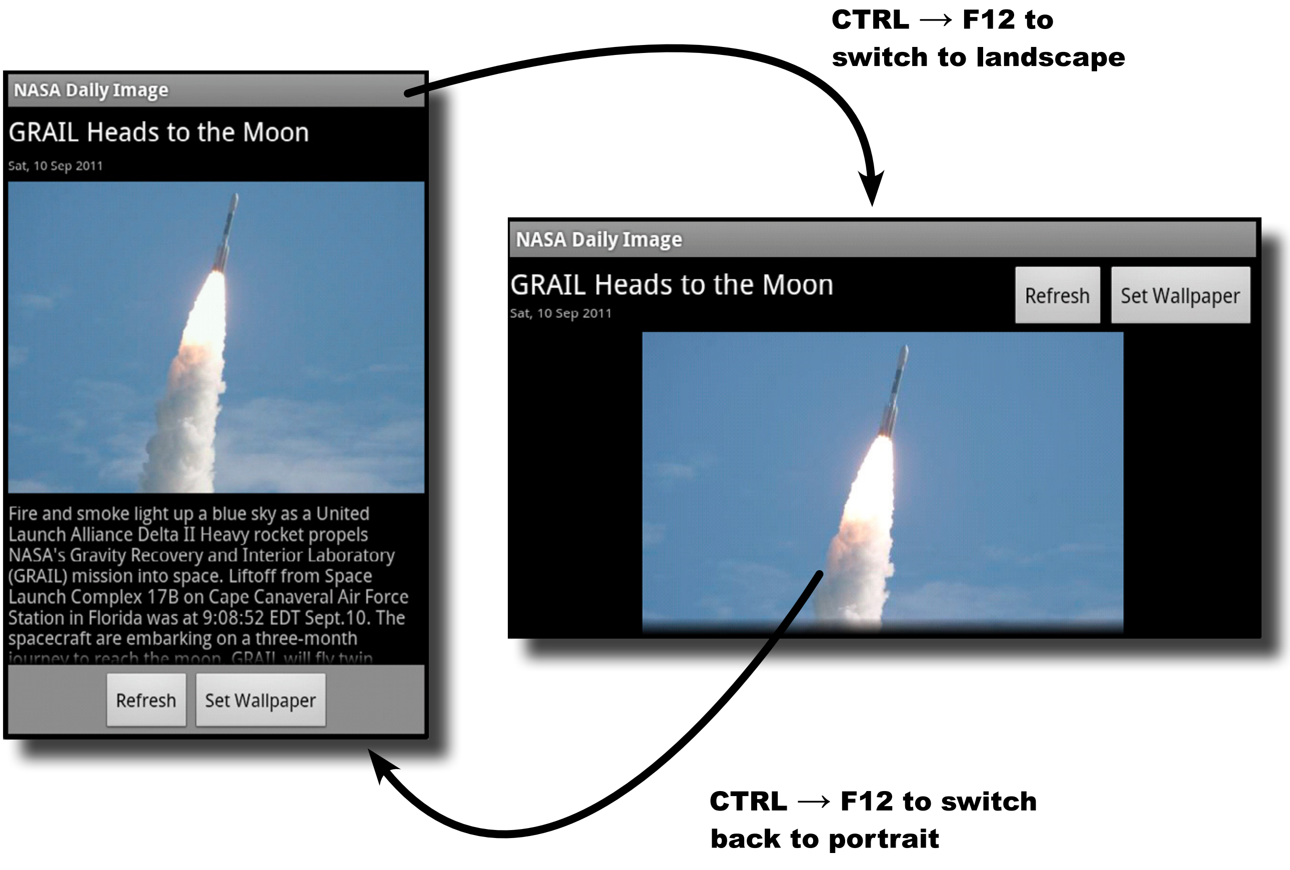
Here is a horizontal LinearLayout buttons on the right.

Add the buttons to the layout.

End of the entire header layout.

Test Drive

Update the layout in your project to match the code updates you did with the magnets. Now run the app again. The emulator will start off in portrait mode. Press CTRL → F12 to switch to Landscape mode and back.





This was a cool improvement for landscape mode.

That button bar looks great in portrait mode but sure was a huge waste of space in landscape mode. With those buttons moved to the top right, you can see almost all of the image, even with the minimal screen height. And with this change, portrait mode is left alone and just the landscape mode was altered. Super cool!

there are no Dumb Questions

Q: I would have laid out this screen differently. Is this the only way to solve this button issue?

A: There are many different ways to have solved this design issue. This is pretty common when you're dealing with user interface design.

Q: What is another way you might have solved this?

A: You'll learn about Android menus in a few chapters. These are actions that are hidden until you press the menu button. Menus are often a good choice if you want to hide functionality but still allow it to be used.

Q: This landscape mode change is pretty minimal. Can I make bigger changes?

A: You can change the screen all around and have entirely different functionality! That said, you probably want to keep landscape mode and portrait mode pretty similar since they are the same screen from your users perspective and they might go back and forth as they move their phone around. Also, remember that the underlying Activity is the same for both landscape and portrait mode, so any features added to either orientation need to be supported by the same Activity.

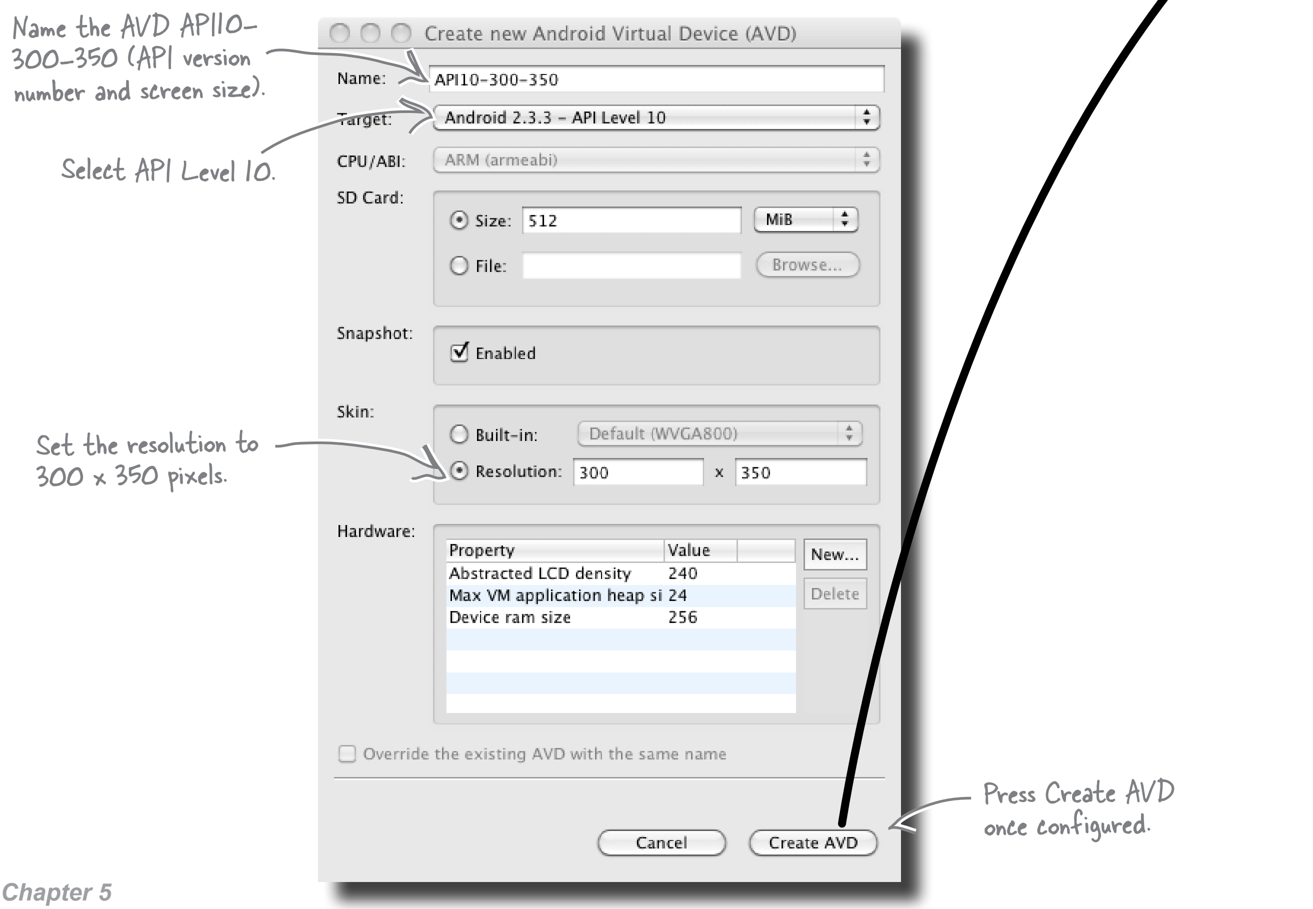
What about super small screen devices?

Now that the landscape mode is taken care of it's time to move on to small screen devices. But as with any other issue, the first step is always to replicate it in your local development environment. Testing landscape mode was easy! All you had to do was switch the orientation of the running emulator. But how do I make the emulator device smaller?

Create an AVD for a smaller screen device

The whole point of creating an AVD (which as a quick refresher stands for Android Virtual Device) is to be able to run an Android emulator mimicking a hardware device. Switching between landscape and portrait mode worked on the same device, but making a smaller screen requires a new device.

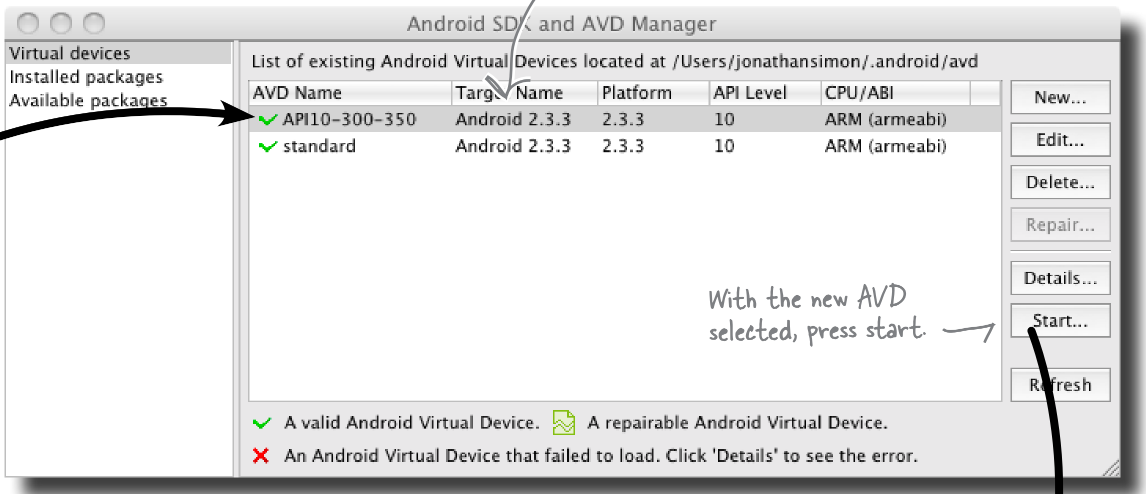
Making a new AVD is easy to do though. Go to Window → Android SDK and AVD Manager. Select Virtual Devices and press "New...".



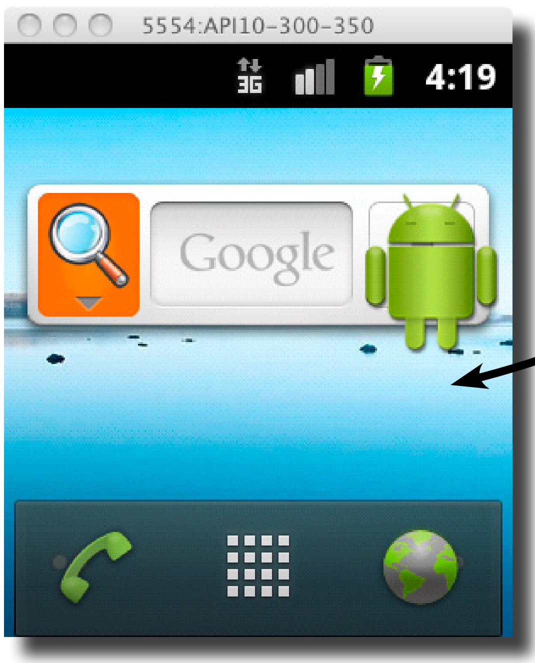
Launch your new AVD

The AVD is just a description of a device. Before you can test your app on that AVD, you need to start it.

After creating your new AVD, you'll see it appear in the Virtual Devices list. Select the new AVD from the list.



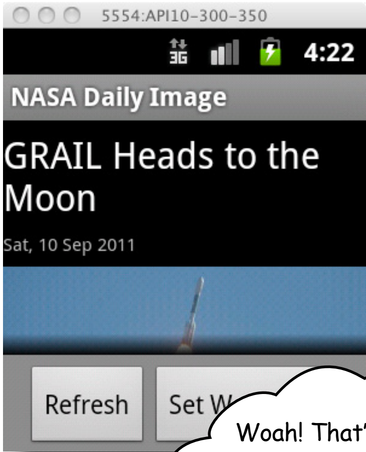
The new AVD is now running.



Run the app on the small device

Now that the new AVD is running you can run the app on it just like you would your original AVD. Run your project now and you'll see it running on the smaller emulator.

Here is the NASA Daily Image app running on the small device emualtor!



Relax

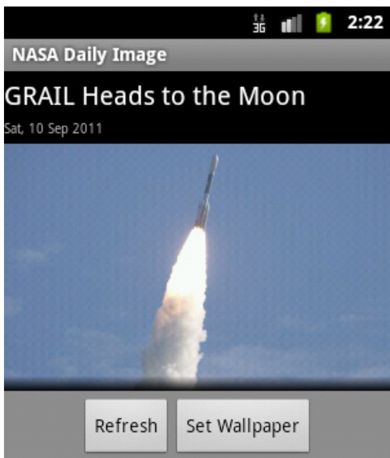
You may have to select the emulator after trying to tun your app.

Your Android development environment knows about the emulators you have running. And if you have more than one emulator running, it will ask you which emulator you want to install and run your app on. If you closed your original emulator before launching the new smaller device, you won't see this.



It looks really different!

There are always going to be little differences between devices and emulators. But there shouldn't be this drastic of a difference in display between them! Let's get to the bottom of this...



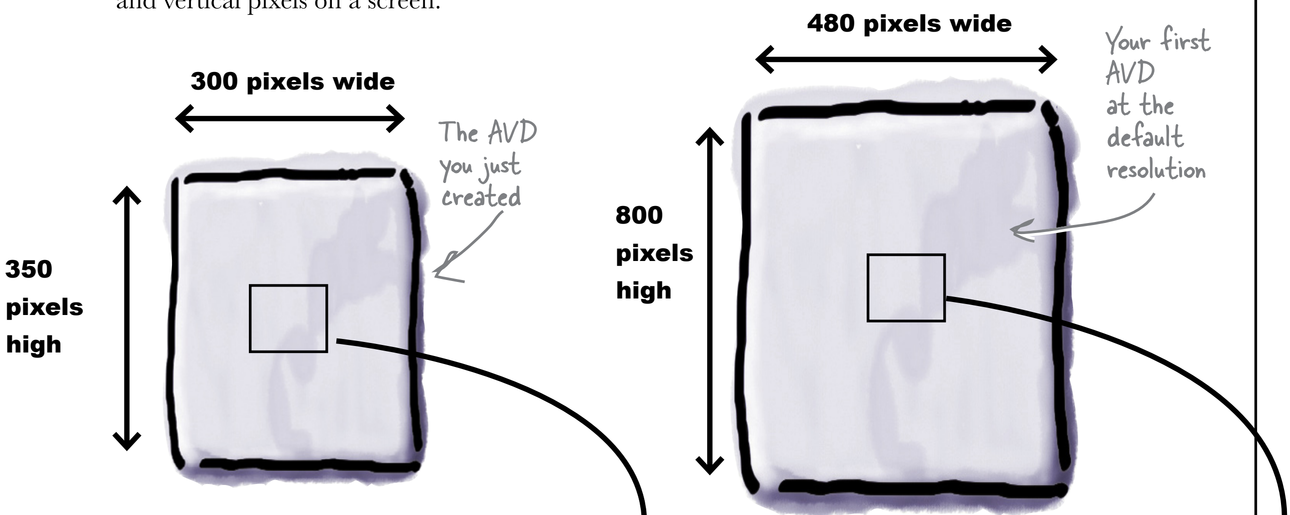
This app looks totally different eben though it's the same sized screen.

The answer lies with pixel density...

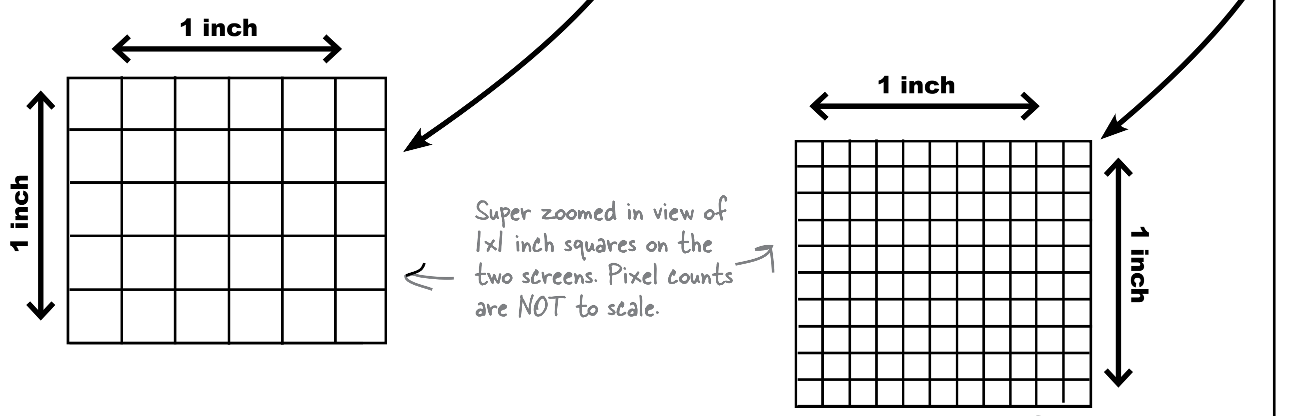
Screens Up Close

There are two screen device properties that effect the way your application looks and runs on a device.

1. **Screen Size**
   This refers to the number of horizontal and vertical pixels on a screen.



2. **Pixel Density**
   This refers to the abstracted number of pixels in an inch.



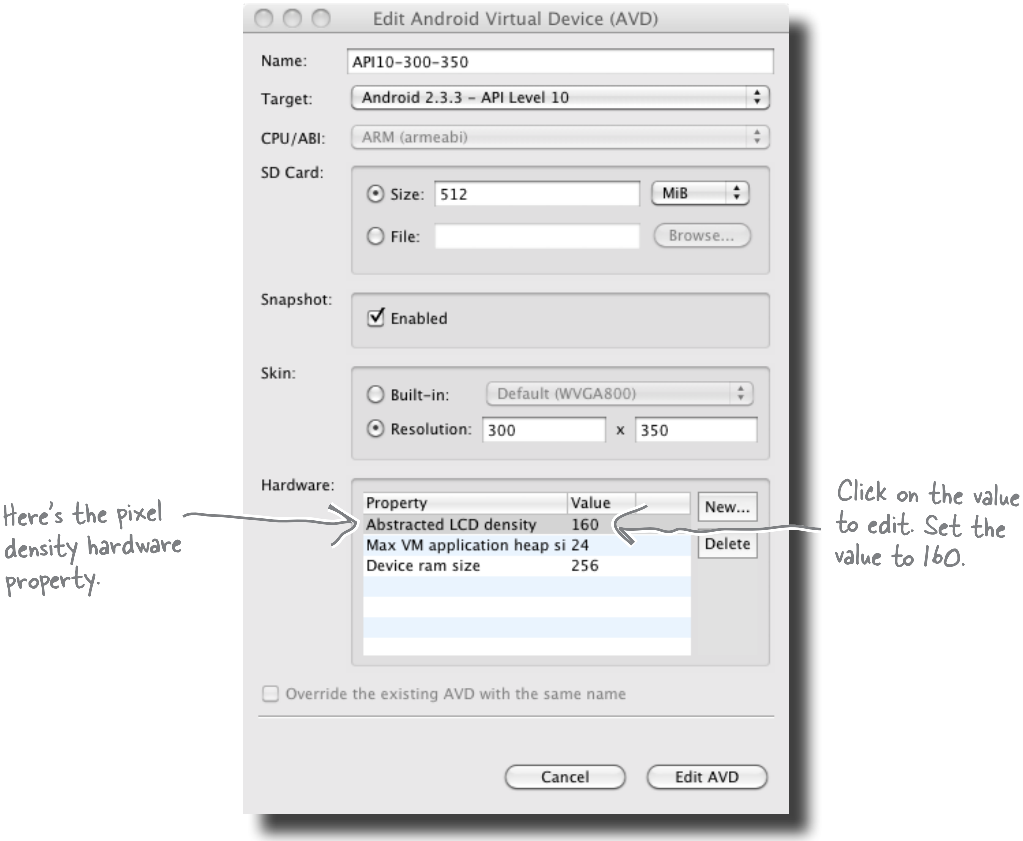
Per inch, the small screen phone actually has twice as many pixels as the big screen.

More pixels over here...

Edit the AVD's pixel density

You can edit your the Pixel Density of the emulator you just configured. Go the Window → Android SDK and AVD Manager and select your new AVD. Click edit and you'll see the same dialog that created your AVD.

Under Hardware, there is a property called Abstracted LCD density. This controls the pixel density of your AVD.



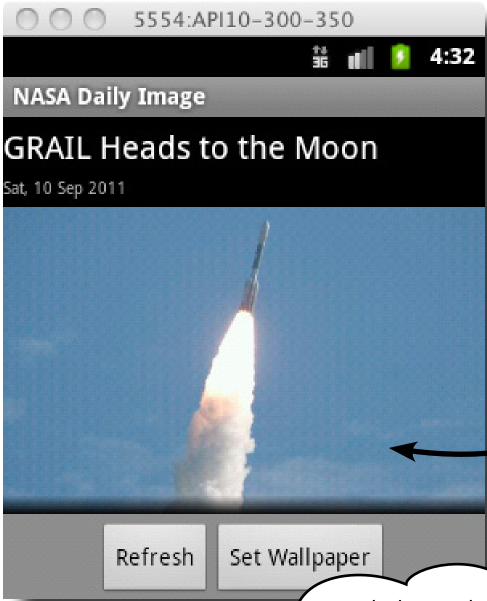


Be sure to use a supported pixel density.

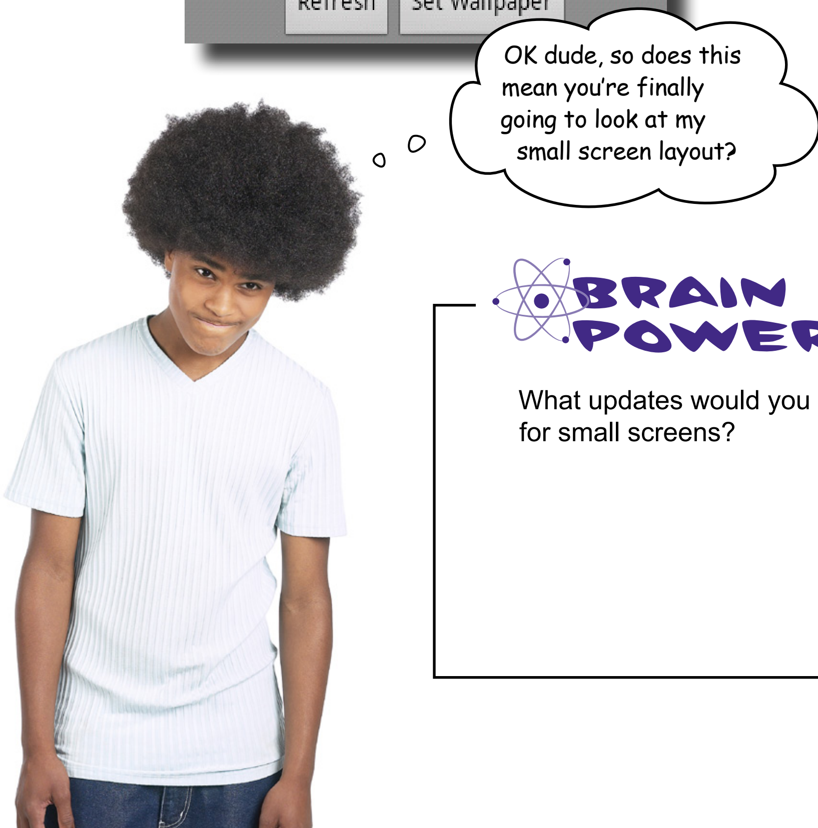
The Abstract LCD Density can only be set to 120, 160, 240, 213, or 320. If you edit your pixel density, you must set it to one of these values.

(Re)start the AVD and the app

Now that you've edited the AVD, close and restart it to make the changes take effect. Once you start the updated AVD, run the app again and see how it looks.



Now the app looks right on the small screen emulator!

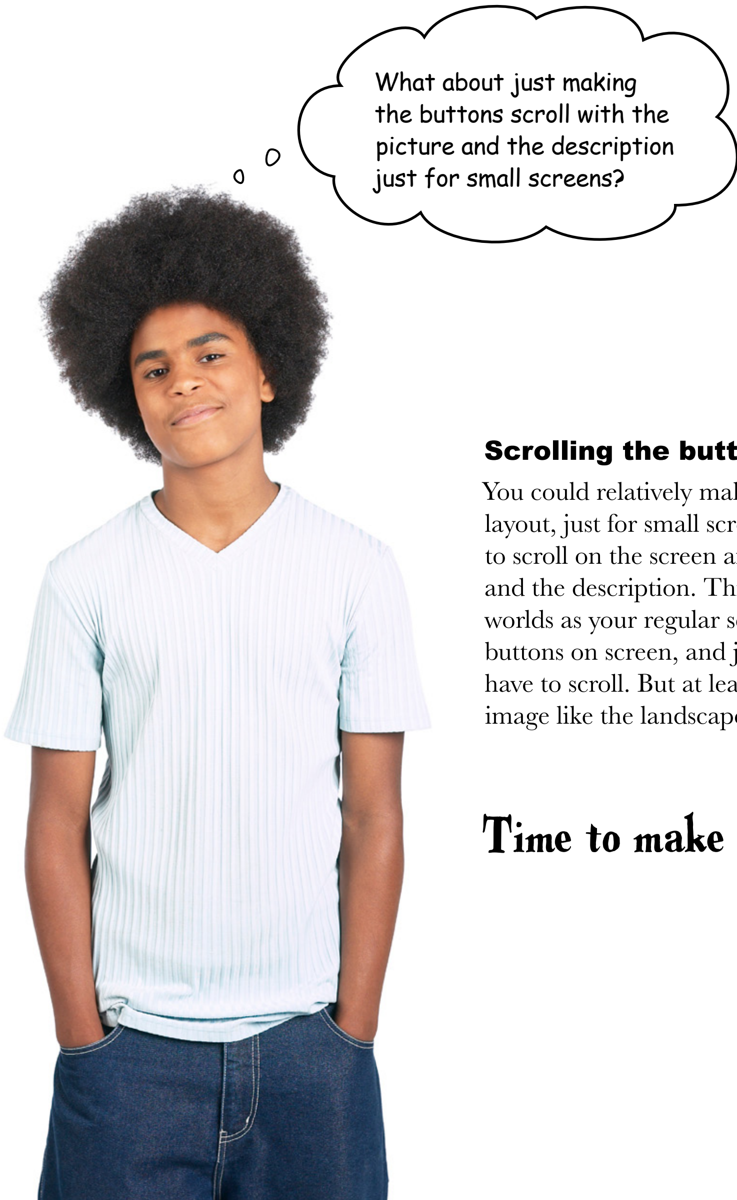


BRAIN POWER

What updates would you make to the layout design for small screens?

Update the design for small screens

Now that you can see what the small screen layout looks like, you can also see that you can't make the same change you did for the landscape layout. Even though they both want to get rid of the buttons, the small screen doesn't have room for the buttons next to the title and date.



Scrolling the buttons is a great idea!

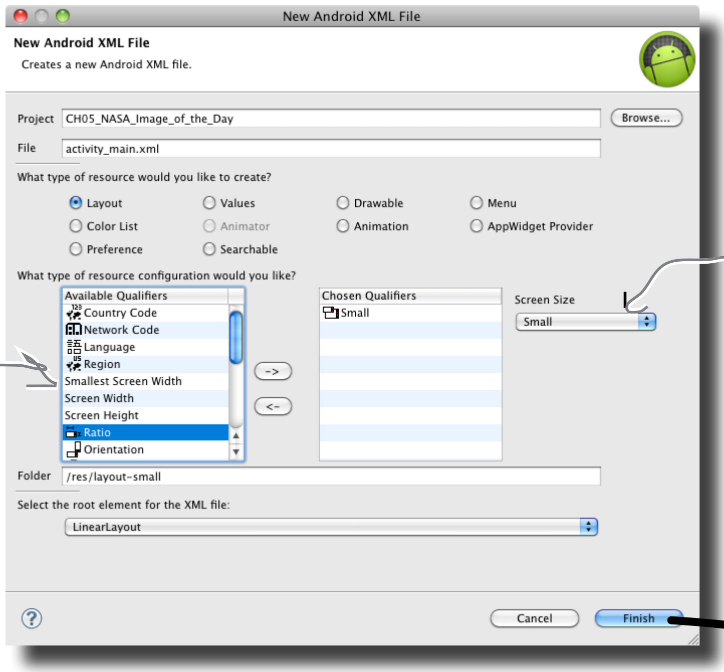
You could relatively make a minor change to the layout, just for small screens, allowing the buttons to scroll on the screen after scrolling past the image and the description. This is a bit of the best-of-both-worlds as your regular screen sizes will still have the buttons on screen, and just the small screens will have to scroll. But at least they'll see more of the image like the landscape layout.

Time to make a new layout...

Create a small screen only layout

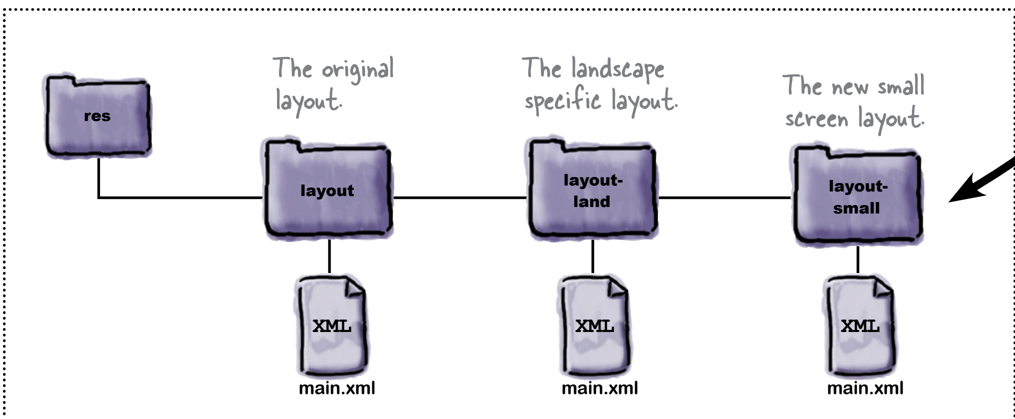
Just like you created a landscape specific layout, you can create a small screen specific layout. Open the new Android XML File Wizard and create a new layout XML file as you did before. But this time, add the size by selecting size from the Available Qualifiers. Once added, select screen size "Small" from the dropdown on the right.

Select size from the Available Qualifiers.



Once added, select small from the dropdown to indicate a small screen size.

When you click Finish, a new layout xml file will be created in the layout-small directory for small screen phones.





Small Screen Layout Magnets

Below are the magnets you need to complete the custom small screen layout. Just like the landscape mode, the small screen layout with the buttons in the `ScrollView` is going to be really similar to the original layout. You just need to recreate the button bar inside the `ScrollView`. Use the magnets below to complete the layout.

Your magnets.

```
<Button android:text="@string/setwallpaper"
        android:onClick="onSetWallpaper"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/setWallpaperButton" />
```

```
</LinearLayout>
```

```
<TextView
        android:id="@+id/imageTitle"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:textSize="20dp"
        android:textColor="@color/image_title_color"
        android:layout_marginTop="5dp"
        android:layout_marginBottom="5dp" />
```

```
<TextView
        android:id="@+id/imageDate"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:textSize="10dp"
        android:layout_marginBottom="5dp" />
```

```
<ScrollView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_weight="1" >
```

```
<ImageView
        android:id="@+id/imageDisplay"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_marginBottom="5dp"
        android:adjustViewBounds="true"/>
```

```
<LinearLayout
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:gravity="center_horizontal" >
```

More magnets.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" >
```

```
</LinearLayout>
```

```
</LinearLayout>
```

```
</ScrollView>
```

```
<TextView
        android:id="@+id/imageDescription"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content" />
```

```
<Button android:text="@string/refresh"
        android:onClick="onRefreshButtonClicked"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/refreshButton" />
```

```
<LinearLayout
        android:orientation="horizontal"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_weight="0"
        android:paddingTop="5dp"
        android:gravity="center_horizontal"
        android:background="#ff8D8D8D" >
```



Small Screen Layout Magnet Solution

Below are the magnets you needed to complete the custom small screen layout. Below are the magnets you need to complete the custom small screen layout. Just like the landscape mode, the small screen layout with the buttons in the `ScrollView` is going to be really similar to the original layout. You just need to recreate the button bar inside the `ScrollView`. Use the magnets below to complete the layout. You should have used the magnets below to complete the layout.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" >
```

The root vertical LinearLayout

```
    <ScrollView
            android:layout_width=”fill_parent”
            android:layout_height=”wrap_content”
            android:layout_weight=”1” >
```

Beginning of the ScrollView.

Vertical LinearLayout as the single ScrollView child.

```
        <LinearLayout
                android:orientation="vertical"
                android:layout_width="fill_parent"
                android:layout_height="wrap_content"
                android:gravity="center_horizontal" >
```

Here are the title and date TextViews as children to the ScrollView's vertical LinearLayout.

```
            <TextView
                    android:id=”@+id/imageTitle”
                    android:layout_width=”fill_parent”
                    android:layout_height=”wrap_content”
                    android:textSize=”20dp”
                    android:textColor=”@color/image_title_color”
                    android:layout_marginTop=”5dp”
                    android:layout_marginBottom=”5dp” />
            <TextView
                    android:id=”@+id/imageDate”
                    android:layout_width=”fill_parent”
                    android:layout_height=”wrap_content”
                    android:textSize=”10dp”
                    android:layout_marginBottom=”5dp” />
```

Here are the image and description Views, also as children to the ScrollView's vertical LinearLayout.

```
<ImageView
        android:id="@+id/imageDisplay"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_marginBottom="5dp"
        android:adjustViewBounds="true"/>
<TextView
        android:id="@+id/imageDescription"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content" />
```

This is the start of the button panel directly inline as a child of the ScrollViews vertical LinearLayout.child

```
<LinearLayout
        android:orientation="horizontal"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_weight="0"
        android:paddingTop="5dp"
        android:gravity="center_horizontal"
        android:background="#ff8D8D8D" >
```

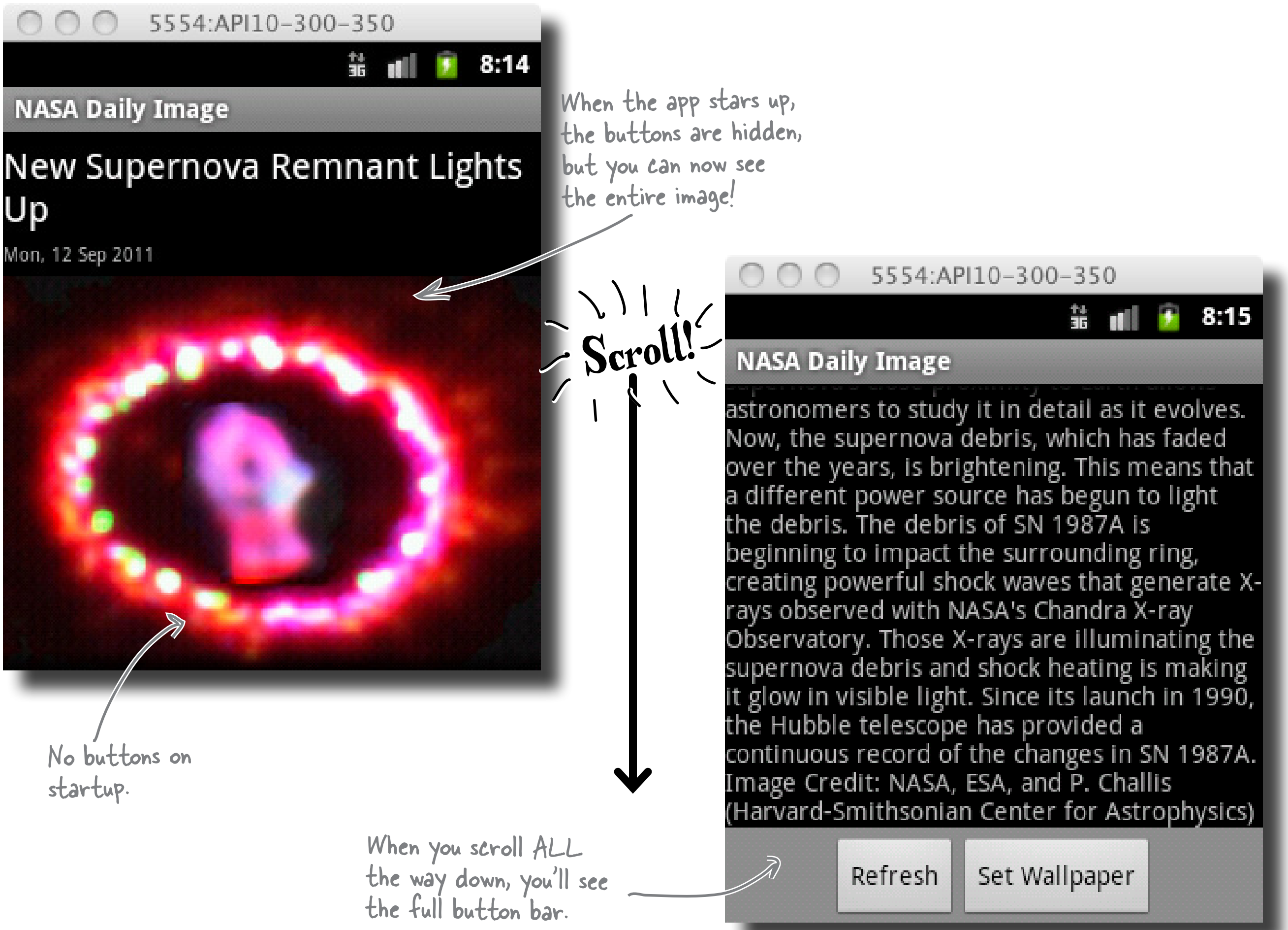
Inside the button LinearLayout is the same, both the refresh and set wallpaper button are added to the horizontal LinearLayout.

```
    <Button android:text="@string/refresh"
            android:onClick="onRefreshButtonClicked"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:id="@+id/refreshButton" />
    <Button android:text="@string/setwallpaper"
            android:onClick="onSetWallpaper"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:id="@+id/setWallpaperButton" />
</LinearLayout>
      </LinearLayout>
    </ScrollView>
</LinearLayout>
```



Test Drive

Now that you have the layout customized for small screens, run the app and make sure your layout changes worked.



The small screen updates look great!

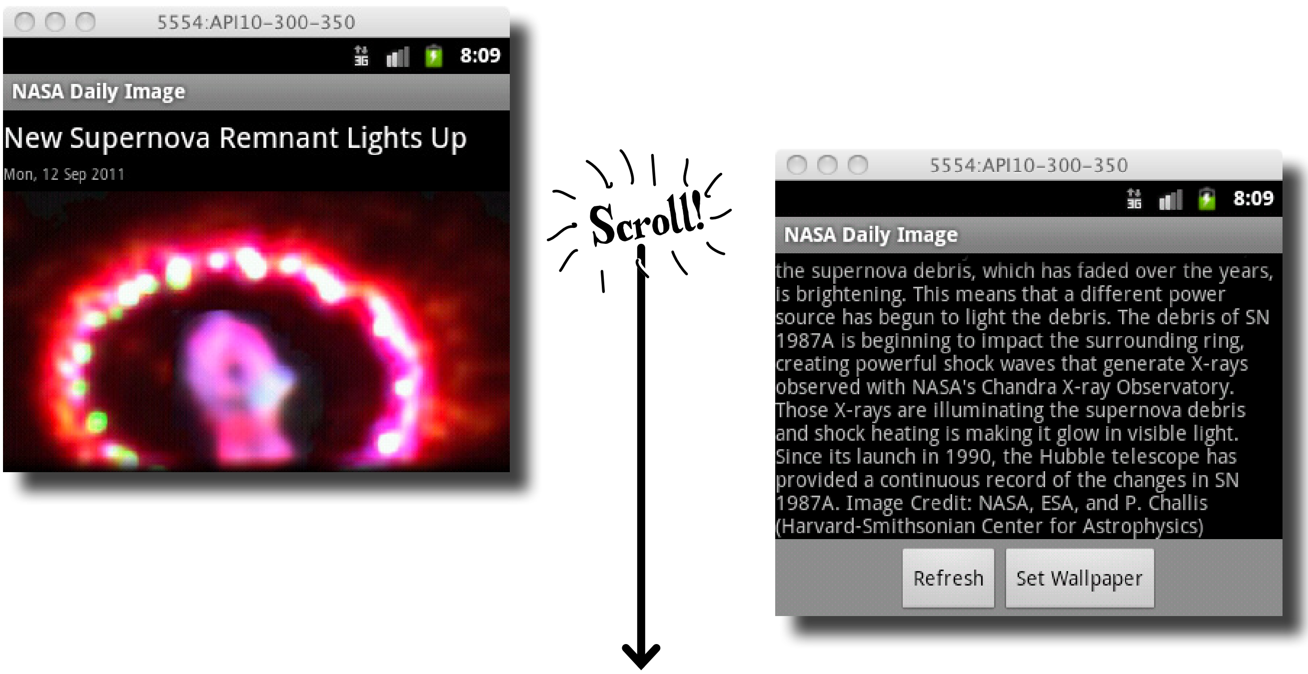
What about small screen landscape mode?

You've put a lot of effort now into customizing the app for landscape mode, and small screens. All of this because you *really* want to make the app the best on all of these different devices! But so far, the issues you knew about were raised by your users. But it's your job as the Android expert to **think ahead for your users and anticipate these layout changes**.

With that in mind, take a closer look at the small screen device again. You customized the main layout, the landscape layout and the small screen layout.

But what about small screen landscape mode?

Turn the emulator into landscape mode (by pressing CTRL-F12) and see how it looks.

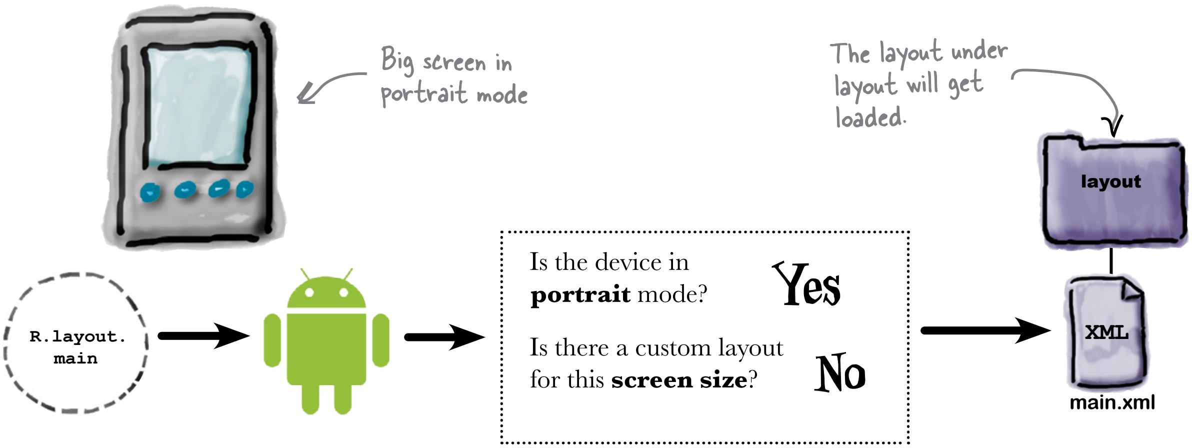


Wait, how did it figure out the landscape small screen layout?

How Android determines the best layout

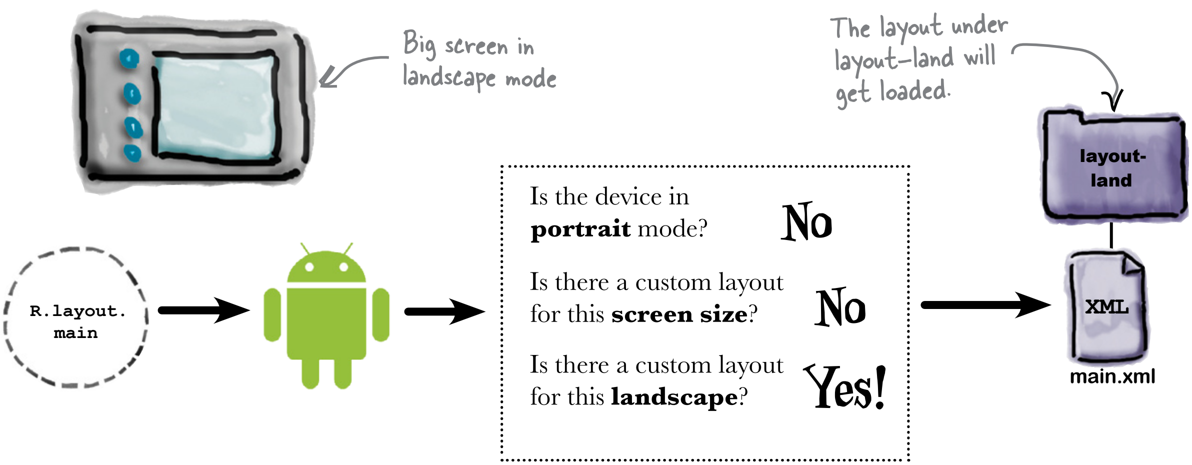
As you're building and customizing your app for multiple screen sizes and configurations, you can end up with a lot of different layouts in your project. It's important to understand which is getting loaded and why. Here's a look at how the four layouts scenarios have loaded their layouts.

1 **A normal device is in portrait mode.**

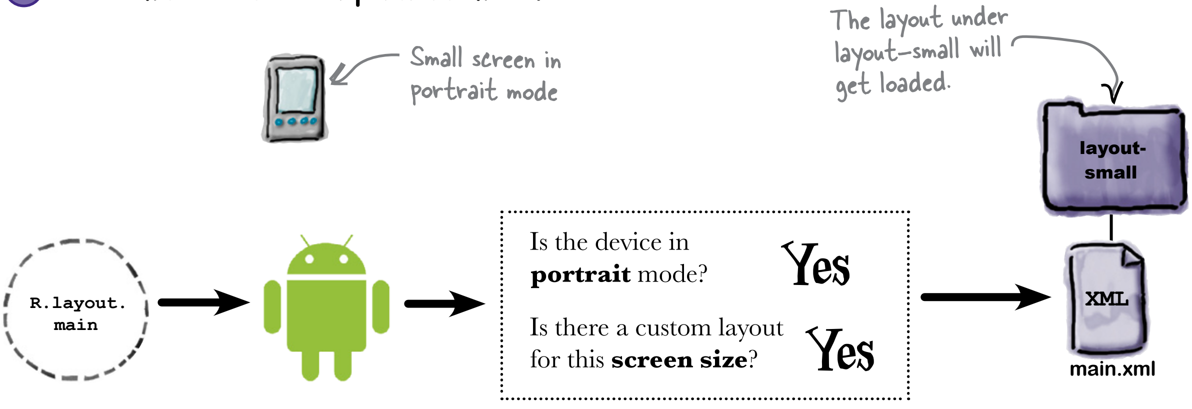


---

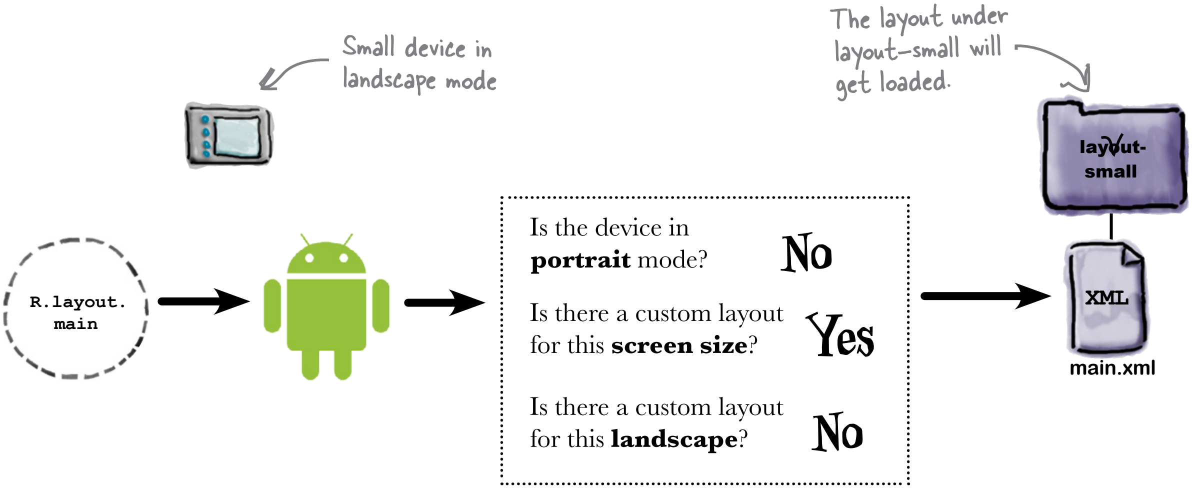
2 **A normal device is in landscape mode.**



3 A small device is in portrait mode.



4 A small device is in landscape mode.





Geek Bits

Check the online docs at http://developer.android.com/guide/practices/screens_support.html for more detailed information on how layouts are selected for other screen sizes not covered here.

Shawn's happy with the app now

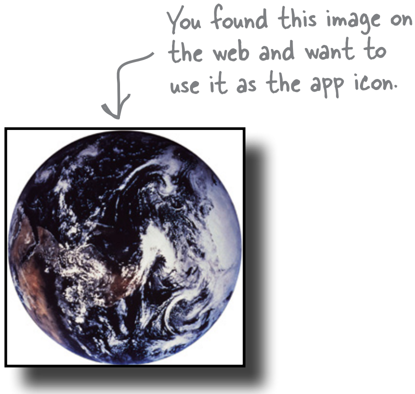
Shawn can see the entire space picture without scrolling (which he's thrilled about). And if he wants to see the description, refresh the app or set my wallpaper, he can always scroll down. You just made a happy user!



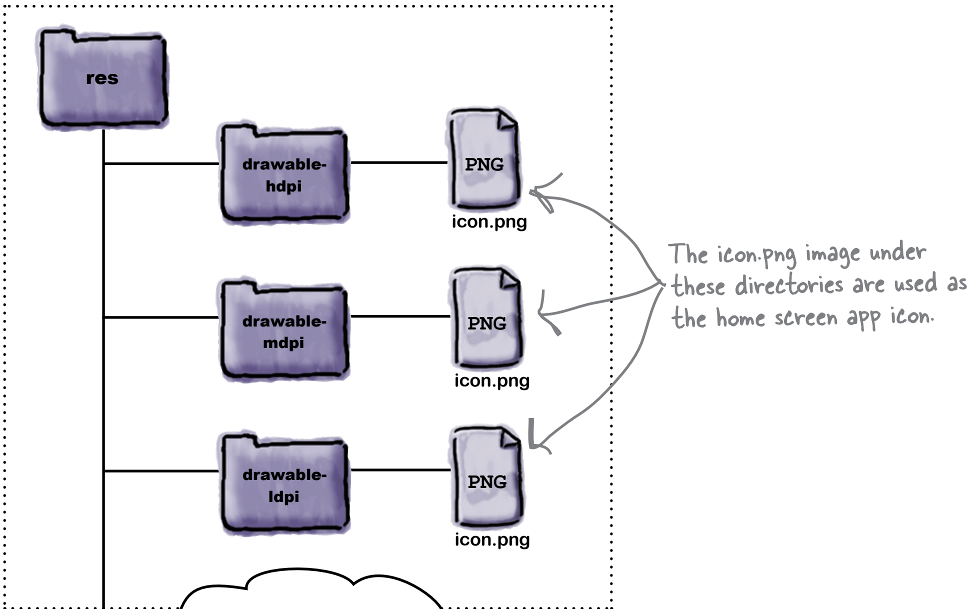
Now, about that icon...

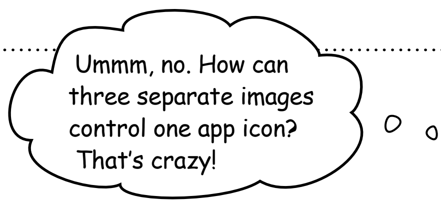
Remember, Shawn did point out one other item that should be fixed before posting the app on the Android Market. He mentioned the app icon was the default icon and it would be a good idea to change it. It will definitely make the app more polished looking to your users, so let's do that now.

After some looking around on the web, you found some free pictures of earth. One in particular looked great for the home screen icon.



As you saw in Chapter 3, app images are stored in the res directory. And the home screen icon is in there in a PNG called **`icon.png`**.







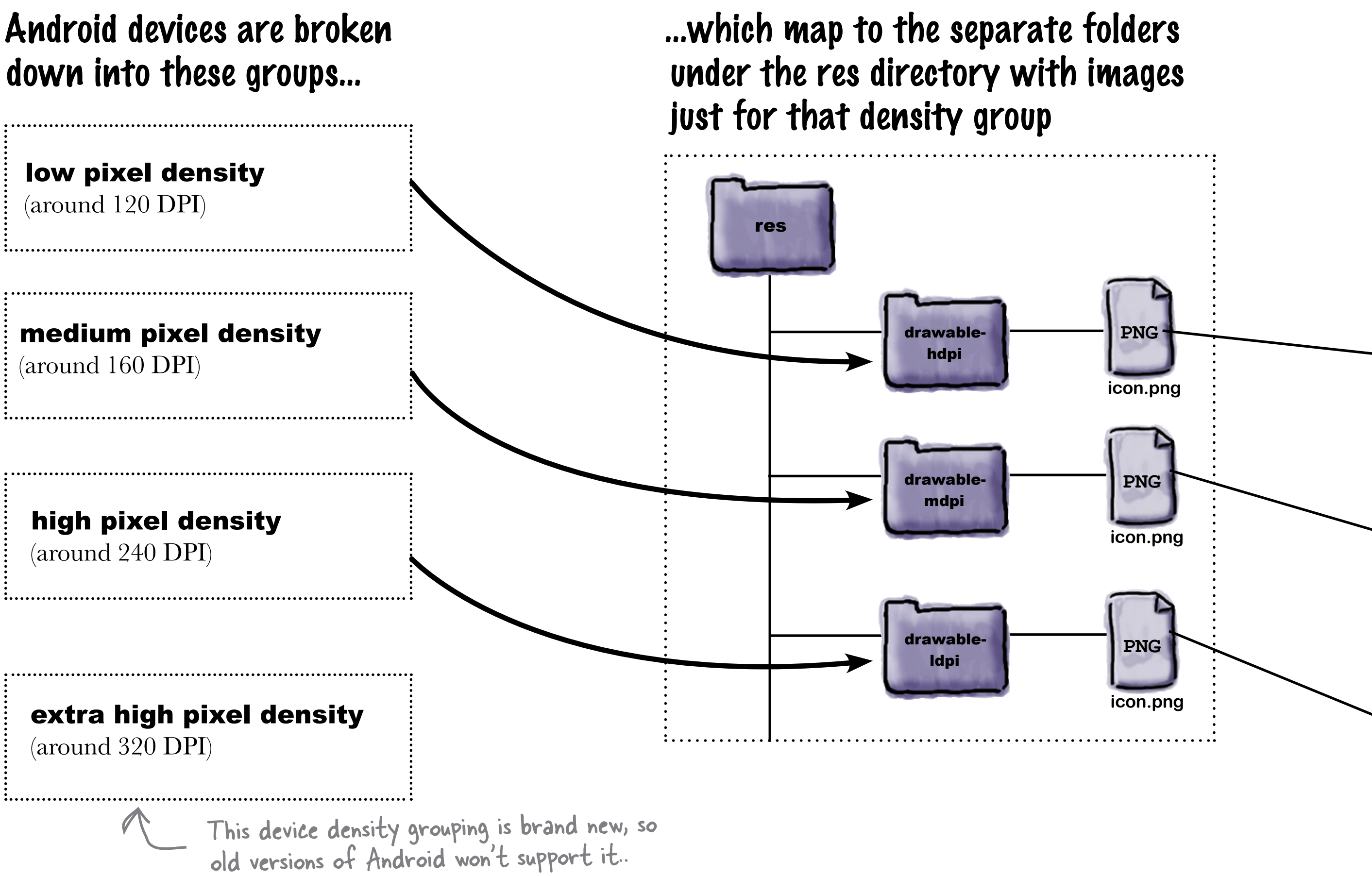
There are multiple images just like there are multiple layouts.

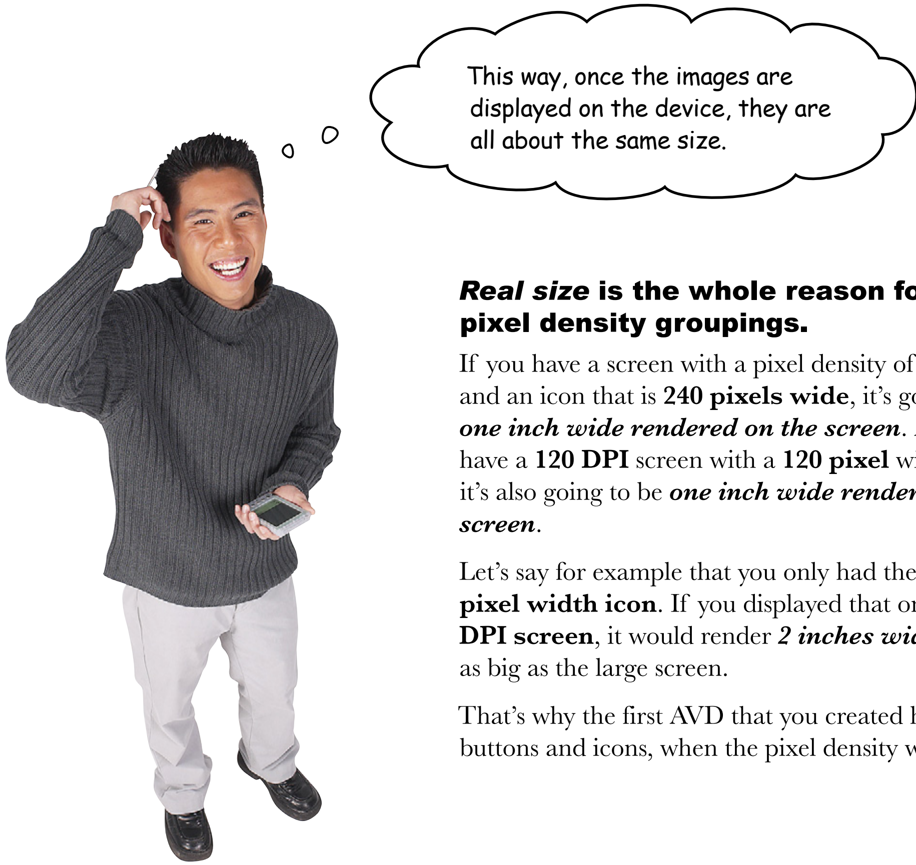
You have to build different layouts to optimize for different screens. And you have to include different images too. Let's see what the different images are optimized for...

Different images for different pixel densities

Earlier in this chapter, the first small screen AVD you created looked weird because the **pixel density was wrong**. Buttons and images were too big and everything looked really squished on the screen. The same weird appearance problems would happen if you use an image that's ***too big*** or ***too small*** for a device's pixel density.

Android solves this problem by breaking devices down into groups of pixel densities (high, medium, and large) and allows you to include images for each group. Then just like the layouts getting chosen at runtime, image resources are dynamically loaded based on the screen size the app is running on.





Real size is the whole reason for the pixel density groupings.

If you have a screen with a pixel density of **240 DPI** and an icon that is **240 pixels wide**, it's going to be ***one inch wide rendered on the screen***. And if you have a **120 DPI** screen with a **120 pixel** wide image, it's also going to be ***one inch wide rendered on the screen***.

Let's say for example that you only had the large **240 pixel width icon**. If you displayed that on the **120 DPI screen**, it would render ***2 inches wide!*** Twice as big as the large screen.

That's why the first AVD that you created had suck big buttons and icons, when the pixel density was wrong.

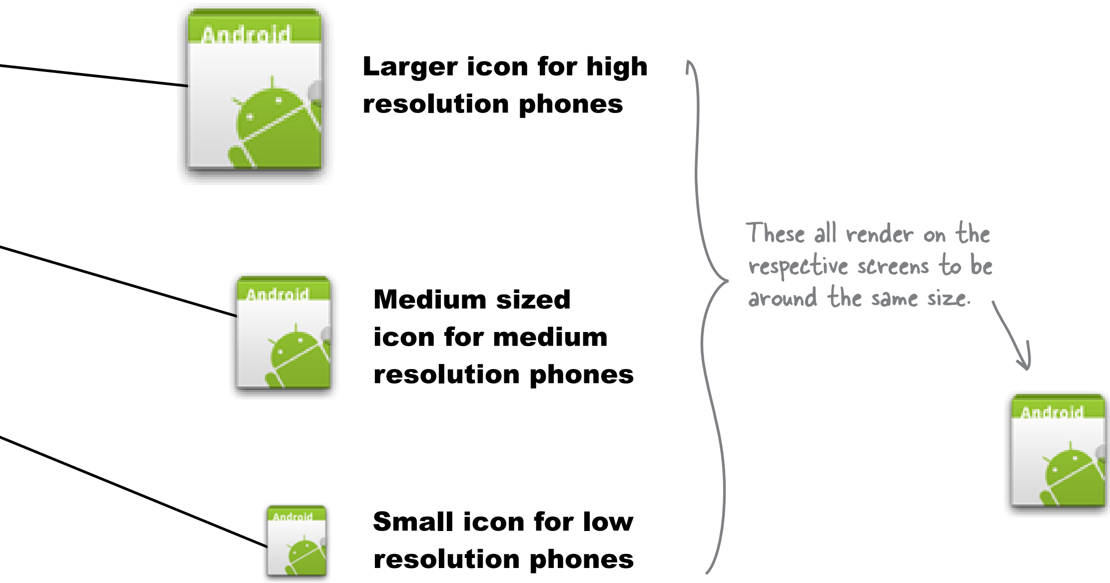


Image standards

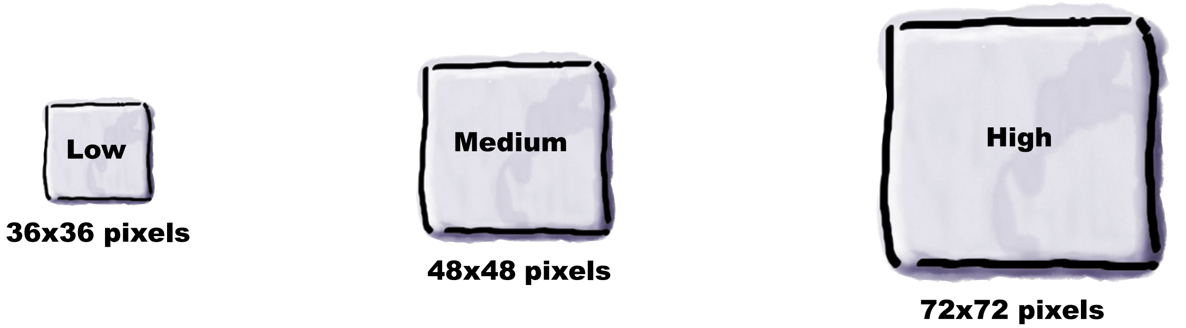
As you're quickly learning. with all of the different devices, there are lots of variations in screen sized and pixel densities. Android divides these up into manageable groups to make things easier. But that's not enough to make a consistent look and feel.

To solve this problem, Android has a published set of guidelines that encourage standards. One of these standards is the image size of the home screen icon.

Do this!

Visit `http://developer.android.com/guide/practices/ui_guidelines/index.html` and familiarize yourself with the Android UI design guidelines.

The guidelines define pixel dimensions for launcher icons at each pixel density.



there are no Dumb Questions

Q: Are there any other design requirements for the icons?

A: The icon design guidelines list a number of other design attributes to use for your home icons. These include recommended margins, colors, drop shadows, and more.

Q: Wow, that sounds like a lot of different requirements. Are there some examples?

A: Absolutely. The icon design guidelines page includes a number of different example icons you can use for reference.

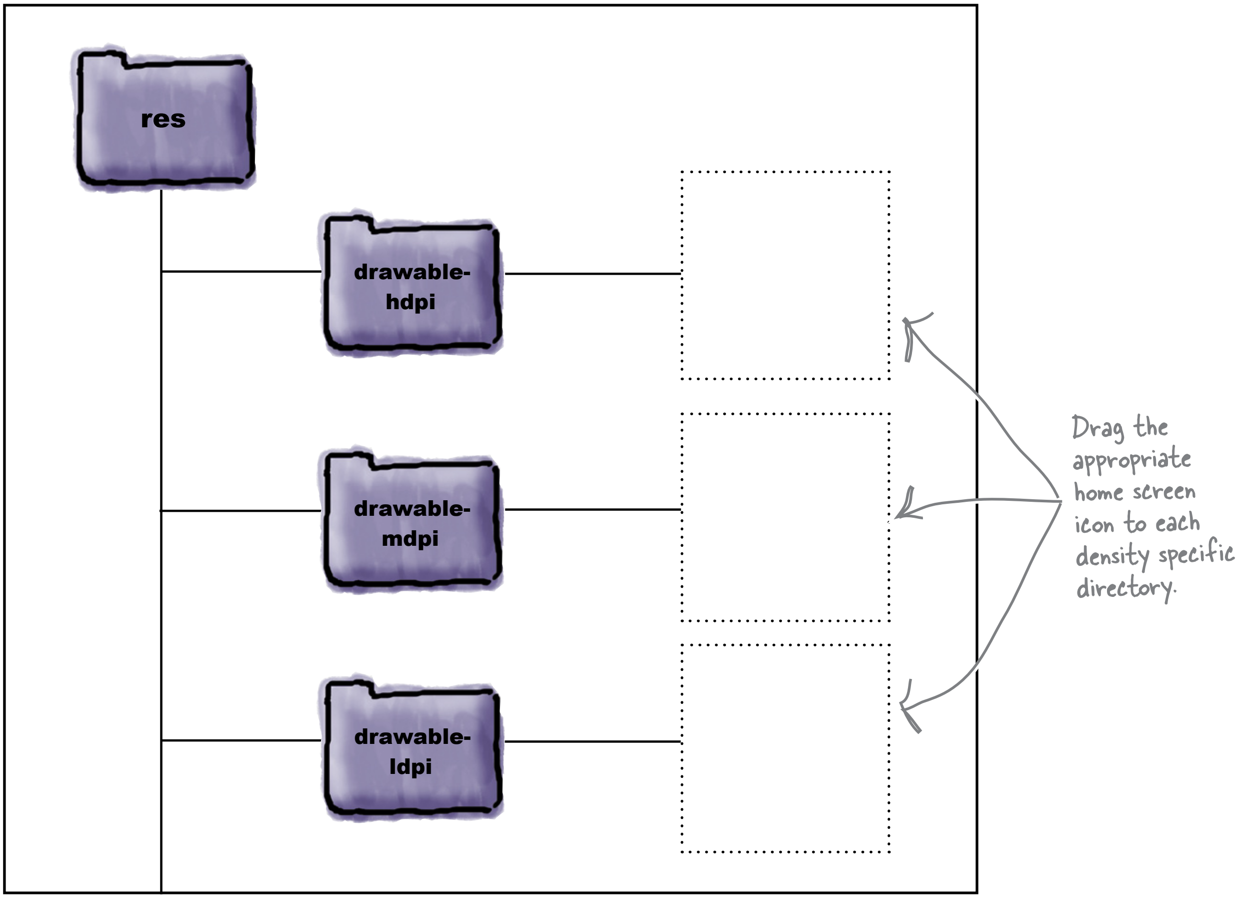
Q: The design requirements for these icons look really complicated! Is it worth it to seek a professional designer's help?

A: Yes. Apps are becoming much more graphics intensive and can often benefit form the help of a professional designer. This is especially true of your launcher icon which will be one of the first things your users see! If you work with a designer, point them over to the UI guidelines page as well as `http://developer.android.com/guide/practices/ui_guidelines/icon_design.html#design-tips` for more information on working with graphics in Android.



Home Icon Magnets

Below are the density specific folders under the res directory. There are magnets for the picture of earth icon resized for each pixel density. Drag the home icons on the squares to the right of the folder they belong to.



This icon is 72x72 pixels.

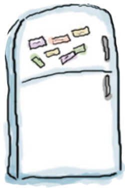


This icon is 36x36 pixels.



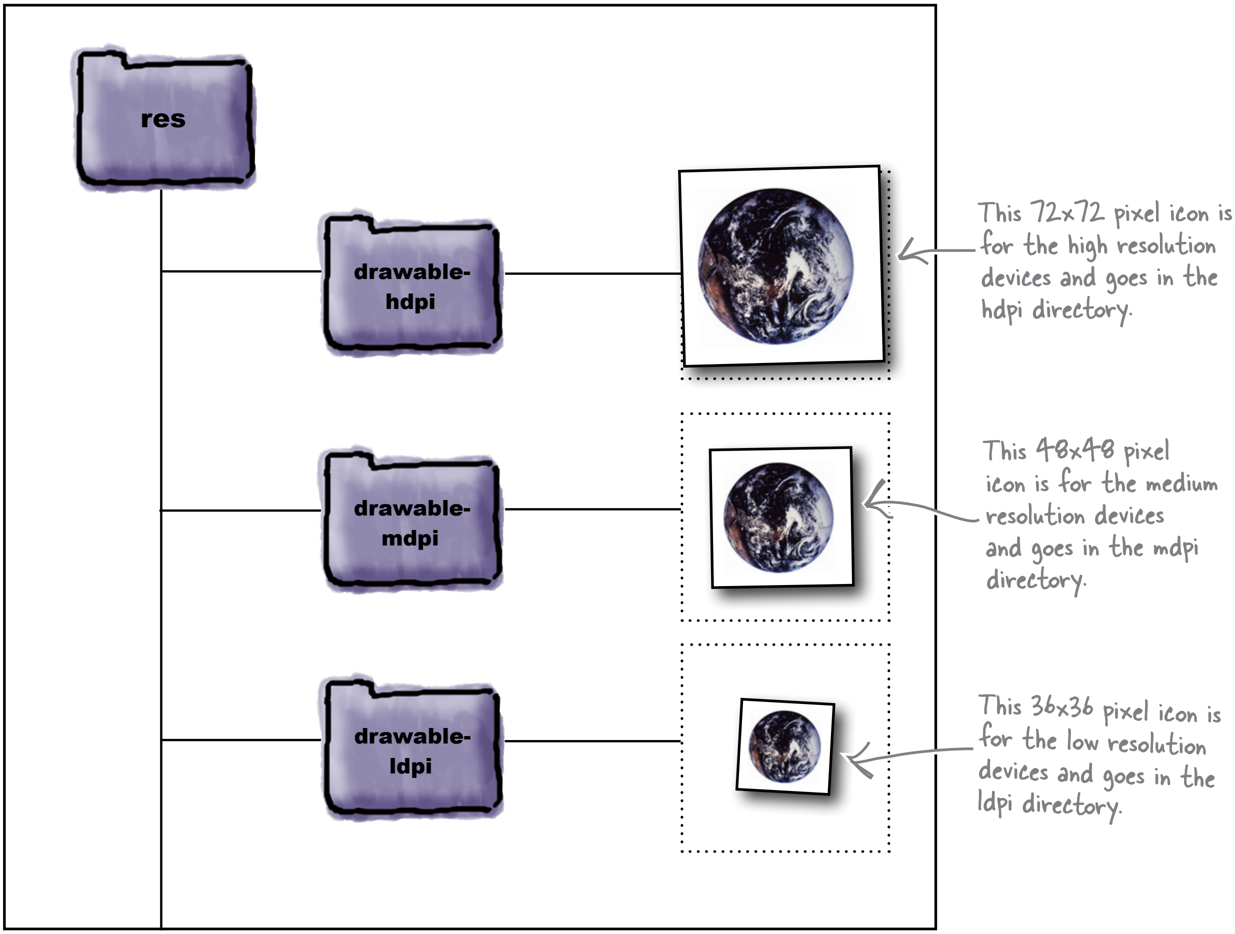
This icon is 48x48 pixels.





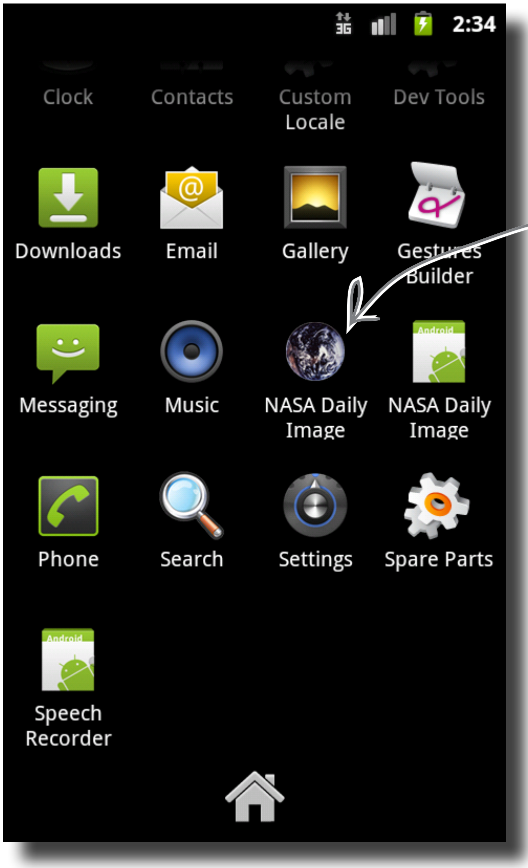
Home Icon Magnet Solution

Below are the density specific folders under the res directory. There are magnets for the picture of earth icon resized for each pixel density. You should have dragged the home icons on the squares to the right of the folder they belong to.



Test Drive

Now that you have optimized icons for each pixel density, run the app in both AVDs and navigate to the home screen. Note the updated icons.



On the large screen high pixel density device, the 72x72 icon is used. But still, it just looks normal.



On the small screen medium pixel density device, the 48x48 icon is used. But again, it just looks normal.

Two screen sizes, two pixel densities and different image to make the images look appropriate on each. Perfection!



It looks like the app is ready for the market

After building the app, tweaking the layouts for screen sizes and orientations, and polishing it off with the home icons, it's ready for people to download and run in from the Market. .

TODO:

Maybe some mention of the market deploy here, or explicityly if we do a chapter 5.5 for deploying to the market...



Go Off Piste

You've done some great work optimizing your layouts for different devices. Here are some additional directions to explore if you're looking for more!

Cover more configurations

This chapter covered one screen size and one orientation customization. Try optimizing the layout for both landscape and portrait on small phones and large phone. Add an additional set of layouts for medium screen sized phones.

Make more AVDs

You currently have one AVD for large screens and one for small. Try creating a few more so you can test your small and large layouts against multiple different sized AVDs. Watch your layout managers dynamically resize based on screen size!

Rearrange the screens for orientation changes

You made a small change between orientations, moving the buttons to a better location. But think of some extreme changes you could make that would benefit orientation differences. Think about adding features or drastic layout differences between orientations. Think about how this will effect the user experience. Is it beneficial or a distraction? Also, think about how the Activity might need to change if you have functionality in one orientation but not another.



Your Screen Toolbox

Now that you've optimized your app for different screen sizes, orientations and densities, you can make all your apps look great across multiple Android devices!

Screen Size

- small screens are at least 426dpx320dp
- normal screens are at least 470dpx320dp
- large screens are at least 640dpx480dp
- xlarge screens are at least 960dpx720dp

** dp is denisty independent pixels **

Pixel Density

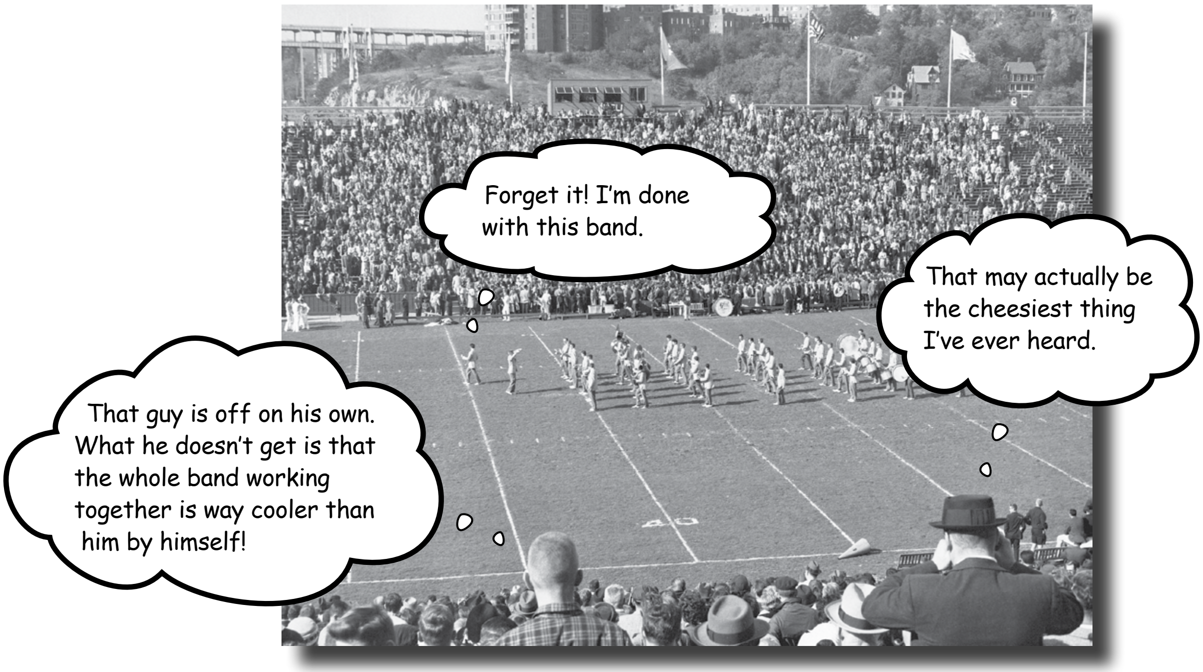
- ldpi is around 120dpi
- mdpi is around 160dpi
- hdpi is around 240dpi
- xhdpi is around 320dpi

** dpi is dots per inch **

- Create multiple AVDs for different screen sizes.
- Change emulator orientation by pressing CTRL F12.
- Create landscape layouts using the New Android XML file wizard and adding the landscape qualifier.
- Create small, normal, and large screen layouts using the New Android XML file wizard and adding the landscape qualifier.
- You can combine qualifiers and make layouts just for one size and orientation, like small and landscape.
- Adjust the pixel density as you create new AVDs to test the correct resource loading.
- Create custom resources for each pixel density you support.
- You can edit AVDs after you create them to adjust screen size and pixel density. But it's still a good idea to have a few AVDs created with configurations for testing.
- Replace icon.png with a custom icon for your app, noting the specific icon sizes for each pixel density.

6 tablets

Running your apps on big(ger) screens



There are more than just phones in the world of Android devices. In the last chapter, you learned how to customize layouts to target different *phone* screen sizes and device orientations. But now you want to take advantage of some of the other Android devices out there like **tablets**. Some of the same strategies still apply, like creating base layouts and optimizing for screen sizes and orientations, but you'll learn about new features to support tablets. You'll also learn about a cool new feature called **fragments** that allow you to configure, and reconfigure the *content* on the screen based on screen size. Let's get going!.

Bobby wants to run the NASA app on a tablet

Bobby's school is running an experiment and gave everyone in his class an Android tablet. Naturally, the first thing Bobby wanted to do was check and see how his NASA Image of the Day app looks on his brand new tablet!

I've been having a TON of fun with the NASA Image of the Day app. And now that I've got this new tablet, I can wait to see the NASA app running on it!



Install the app on a tablet.

The app is already up on the market, so you can download it from any device... including a tablet! You can also install the app directly on the tablet just like a phone.

Let's see how it looks...

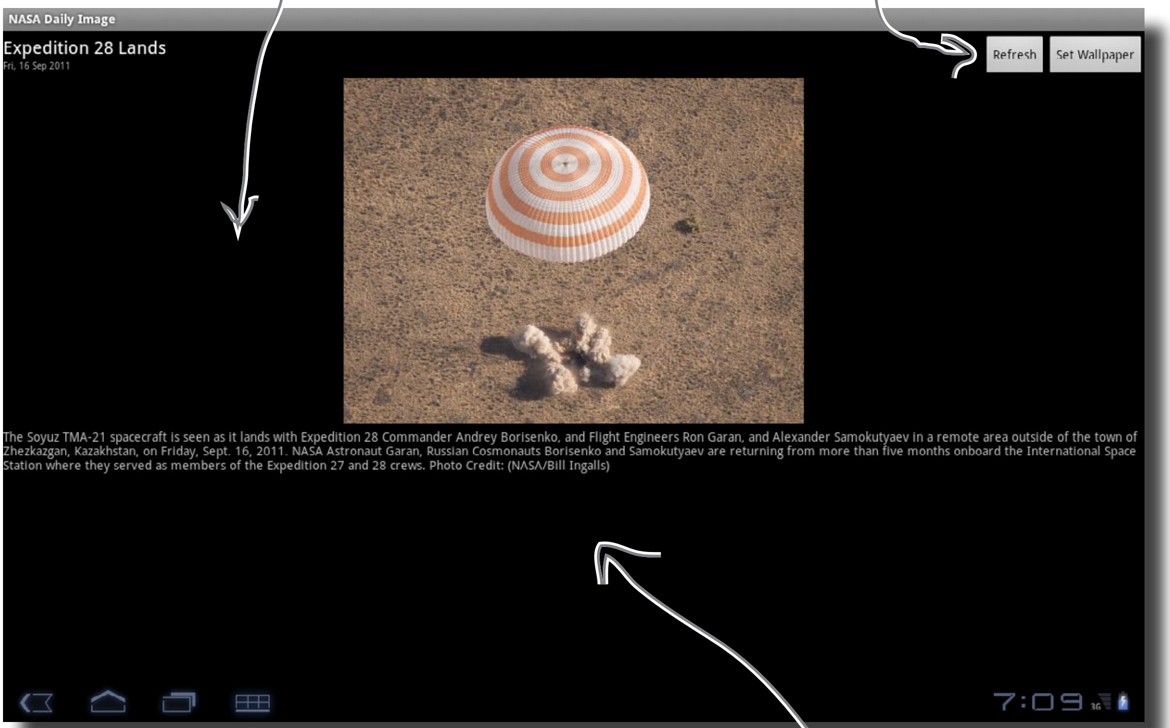
The app doesn't look so good

The title is really small on this huge screen, the image is centered with too much blank space, and the text goes all the way across the screen. It looks bad because we are running a layout that was designed for a phone and running it straight on a tablet.

The title is too small.

There is a very large amount of blank space.

The app loaded the landscape layout, since this is in landscape mode.



Really wide stretched text...

Android tablet users describe apps like this as being humongified!

Since they act like they are running on humongous phones!

What about all that blank space?

Right now, the app has a lot of blank space when it runs on a tablet. Bobby's teacher, Kevin, has an idea for you.

Why don't you fill up some of that free space by displaying NASA educational news?

Adding a second feed sounds like a great way to fill out the app!

Since the school gave out tablets, all of Bobby's classmates have them now. And they all want to run the NASA Daily Image app. While they are looking at the daily image, a NASA feed displaying information specifically oriented around education sounds like a perfect fit. So you're not just adding stuff to fill out the screen, you're going to be **adding additional useful content for your tablet users**.



Bobby's teacher Kevin.

The NASA education news feed activity

Kevin thinks this is such a good idea that he built the Activity for you to display the NASA Educational News feed.

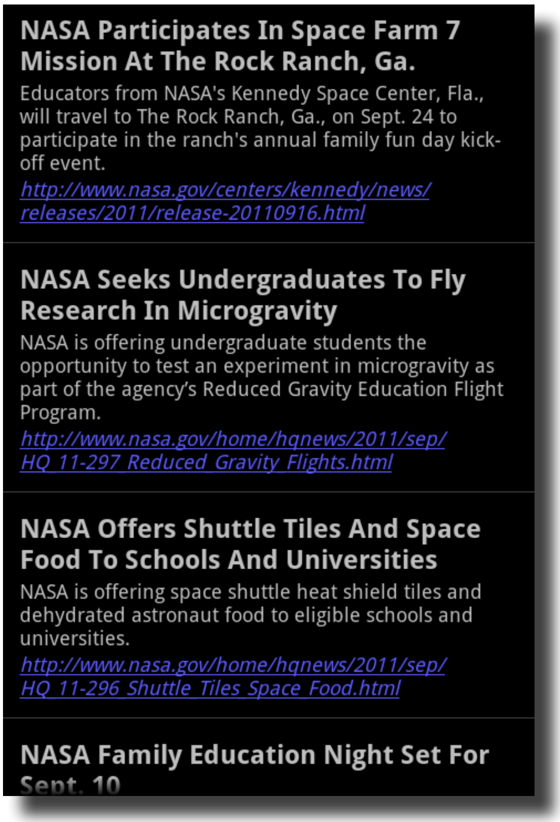
Download the sample code for Chapter 6. There is a project called `CH06_NASA_Image_of_the_Day` that includes a new Activity for displaying the NASA Education News feed in `NasaEdNews.java`.

You'll be taking this code and the code you've written in earlier chapters to make these two Activities work together for a tablet app.

Do this!

Download the sample projects i fyou haven't already. Open the project CH06_NASA_Image_of_the_Day and you'll find a new Activity called NasaEdNews and all of the feed parsing code to make it work.

An activity displaying the NASA education new feed.



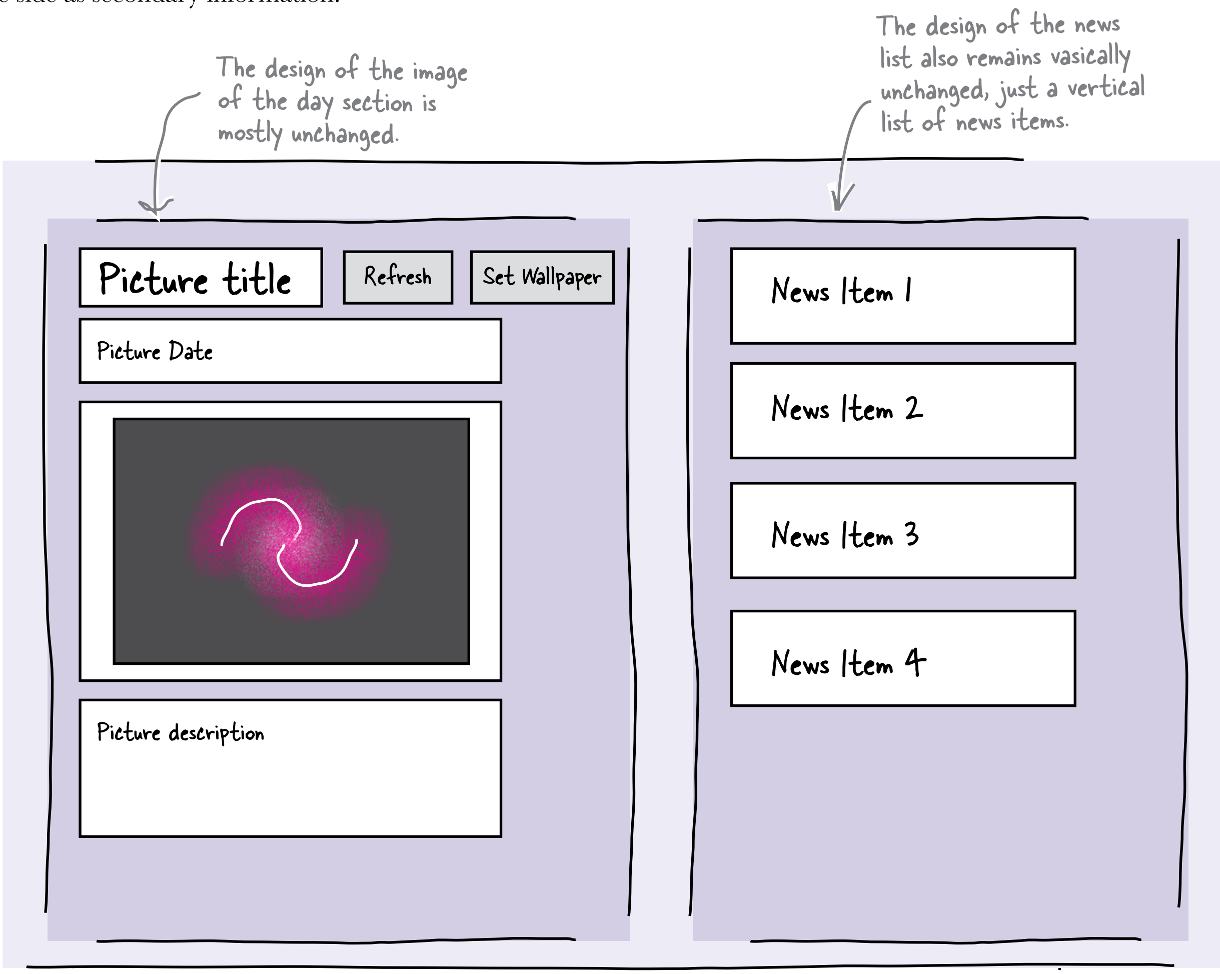


You'll learn all about lists in the next chapter

The NASA Education News Activity is displaying each result in a special View called a `ListView`. You'll learn all about them and how to build and customize your own in the next chapter. For this chapter, you can use the sample code you've downloaded.

How do you want the app to look?

There are a number of different ways to design the app to include both the daily image and educational news. Here is one design that uses space well, and also keeps the news off to the side as secondary information.

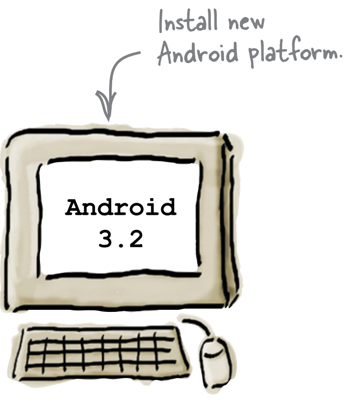


The plan

There are a number of different ways to design the app to include both the daily image and educational news. Here is one design that uses space well, and also keeps the news off to the side as secondary information.

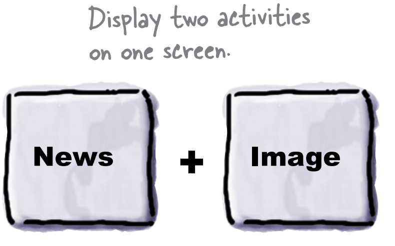
1. Update your development environment

Android tablets are running Android version 3.0 and above. In order to be able to develop tablet specific Android functionality, you'll need to update your development environment to use an Android platform 3.0 or above.



2 Combine the activities

The code for the Nasa Image of the Day and the Education News feed are in two different Activities. These need to be combined into a single Activity to they can be displayed on the screen.



3. Test the new combined activity

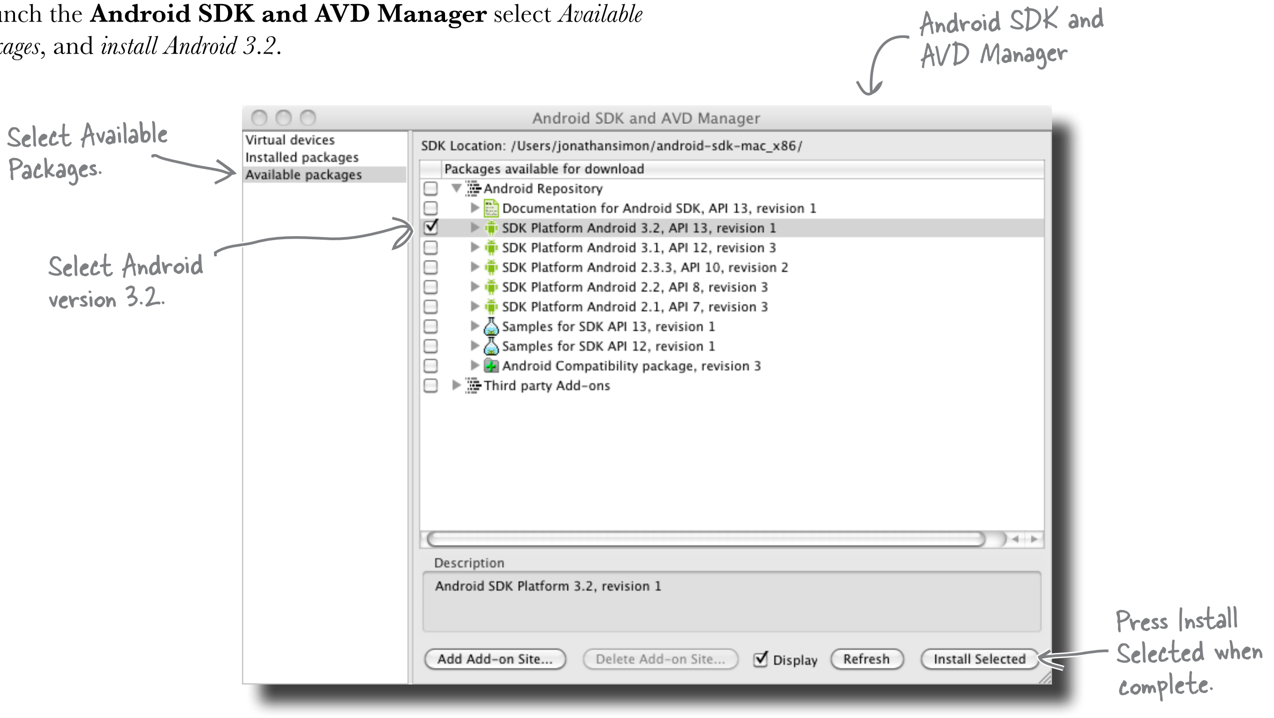
You're going to be moving a lot of code around to make both Activities display on the screen together. As always, you'll need to test your code and make sure nothing is broken (and fix it if it is)!



Add a new platform

In Chapter 1, you installed all of the Android development tools, including the base SDK and one version of the Android platform (Android 2.3.3, API Version 10). Tablets use a later version of Android starting with Android version 3.2 (API version 13).

Android is built to handle these differences by allowing you to install multiple platforms at once in your development environment. To work with Tablet specific functionality, *start by installing Android 3.0*. Launch the **Android SDK and AVD Manager** select *Available Packages*, and *install Android 3.2*.



there are no Dumb Questions

Q: How come Android versions have a version number and an API number?

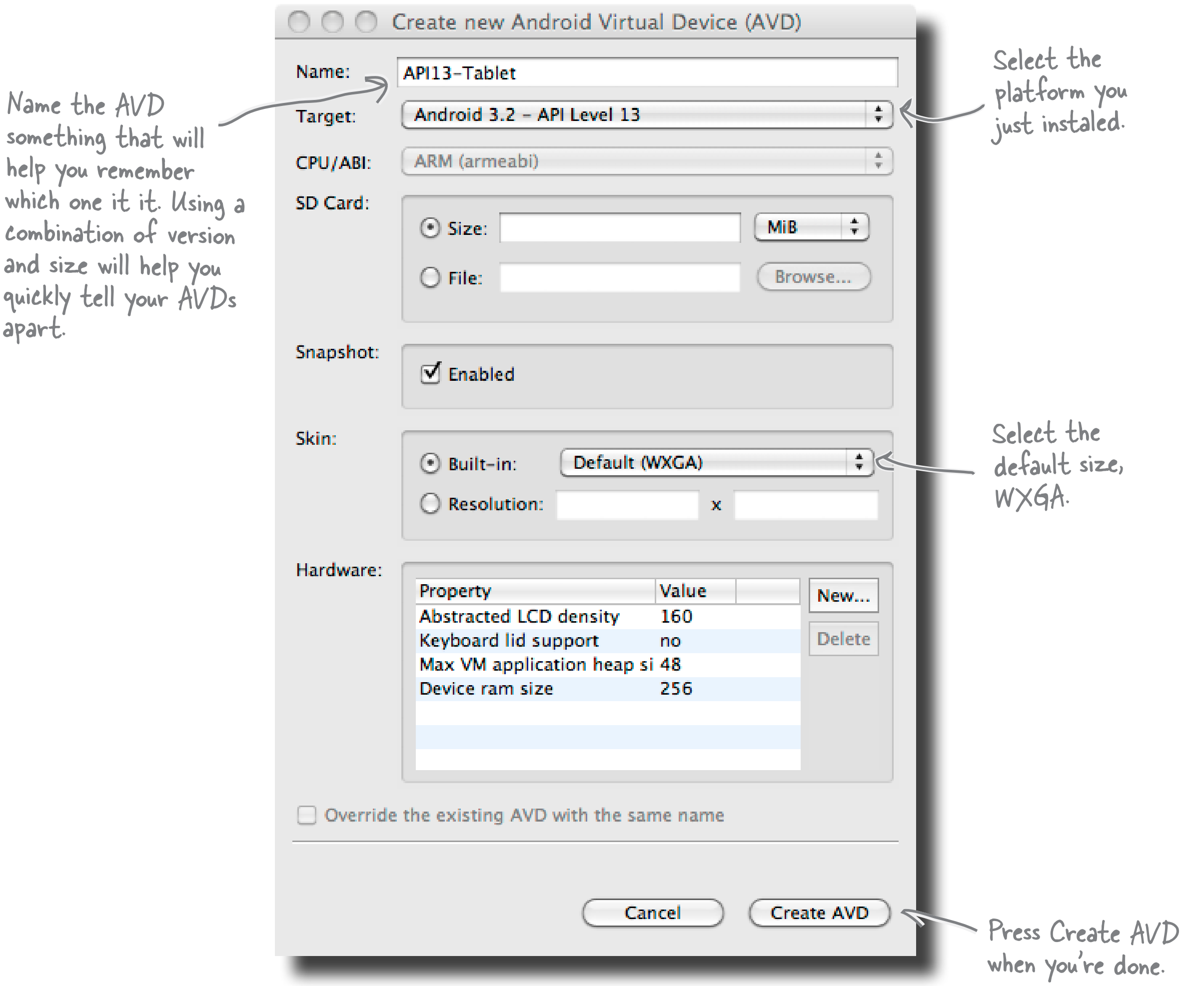
A: Since your app will be running on multiple Android versions simultaneously, the Android platforms are very specific about API changes. And the version number isn't enough to let you know what version the API is. For example, 2.3 is API version 9, 2.3.3 is API version 10, and 3.0 is API version 11. Since the numbering jumped from 2.3.3 to 3.0, the API version helps keep versions in sequence.

Q: How come there are separate versions for Android and Google platforms for each release?

A: The base Android platform versions include the core Android platform. Google provides an extended version of each platform with additional APIs including maps and other cool add ons.

Setup a tablet AVD

Now that you have a new Android platform version installed, you need to setup an Android Virtual Device (AVD) that uses it. Then when you launch that AVD, you'll be running an emulator with the latest version. And in this case, we'll want to setup tablet dimensions since we're testing tablets.



Start the tablet AVD

Once your new tablet AVD is setup, start the AVD by going to Android SDK and AVD Manager → Virtual Devices and launching the new API13-Tablet device.

You'll notice it looks a lot different from Android 2.3.3.

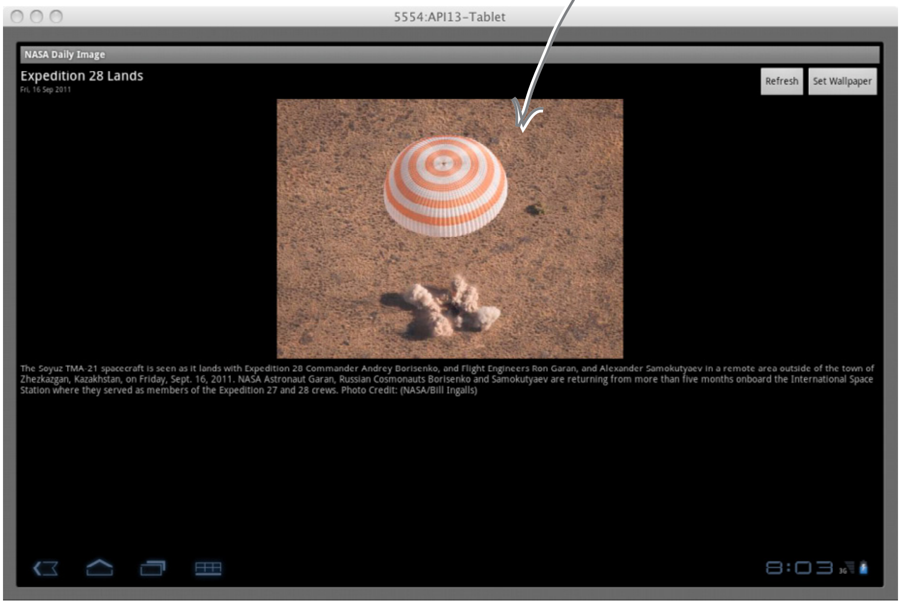


Now let's run the app on the tablet AVD

Run the app on the tablet AVD

Now that the tablet AVD is running, you can run your app on that AVD as you normally would (using the play button or right clicking on the app in Eclipse and selecting run).

The app is running in the emulator.





Android will check where you want to run your app.

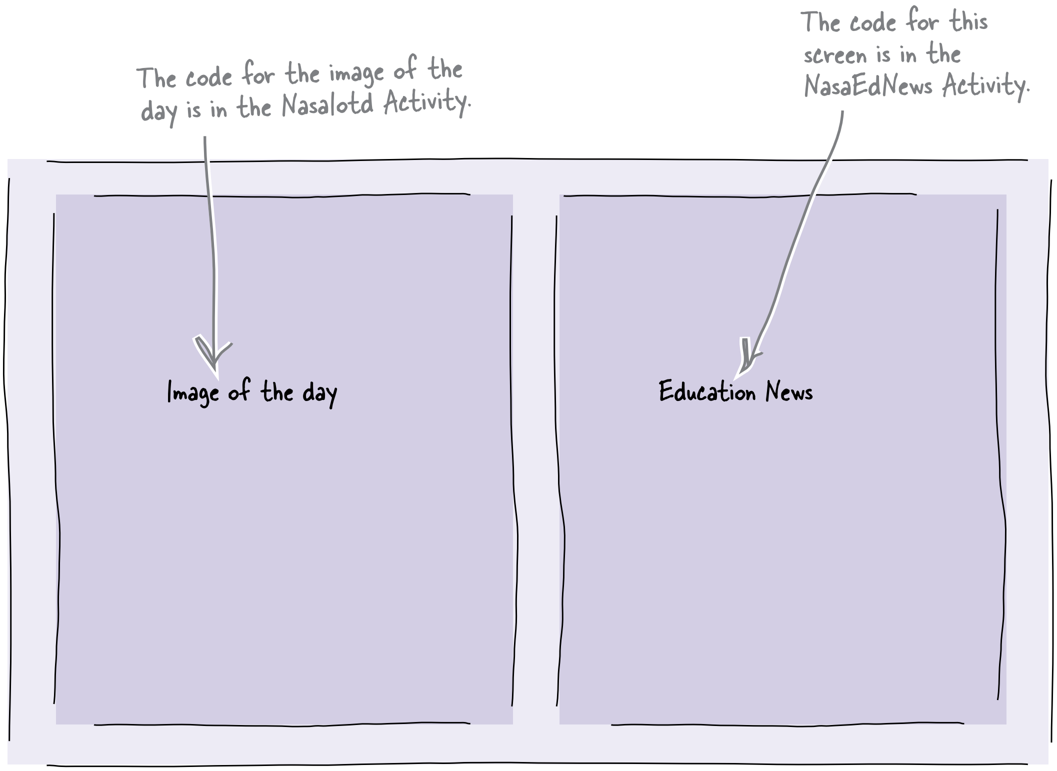
If you have both Android emulators running and you run your app, you'll be shown a dialog with the available devices. You can then choose where you want to run the app from the list of devices.

If you have multiple AVDs running or devices connected, you'll get a dialog like this when you run your app.



Combine the activities

Now that you have the app running in the tablet specific emulator, it's time to start implementing the design changes to combine the activities on one screen.



One combined activity

One way to make this work would be to combine both the Image of the Day and the Education News Feed Activities into one single **combined Activity**. There are a few big downsides to this.

First, you would be duplicating code since you want to be able to keep the Activities separate to run just one at a time on a small screen device. On top of that, right now the code for each function (the image and the news feed) is encapsulated in an Activity. And it would be great to keep them that way.

You could make one combined Activity to display both Activities on one screen...


Wouldn't it be dreamy if you could combine multiple Activities in a single screen without having to combine them into one giant Activity. But I know it's just a fantasy...

Use fragments

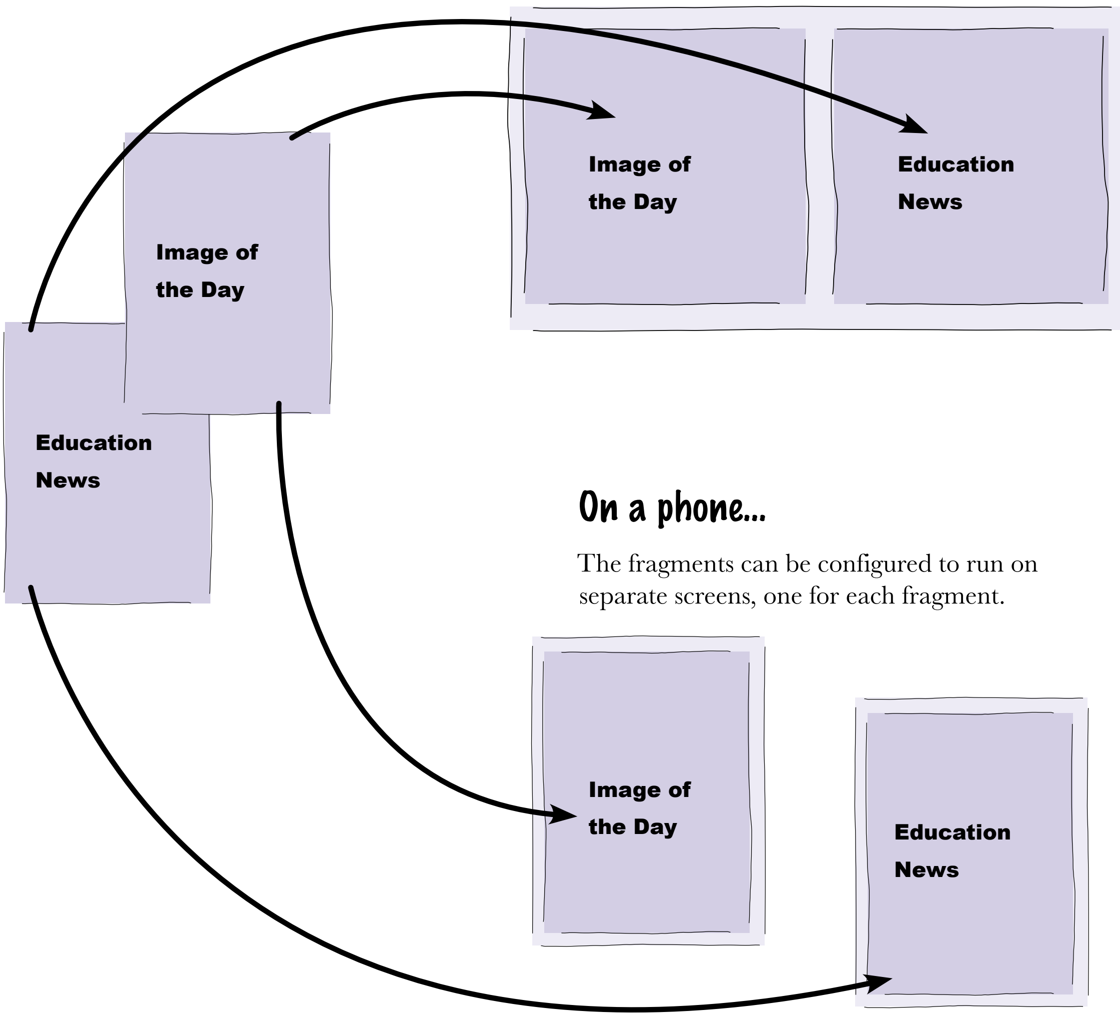
It's a natural progression to add more features per screen once you move to the larger screen sizes of tablets. But since Android devices take on *so many different shapes and sizes*, it's important to remain **flexible** and be able to arbitrarily combine parts of different screens.

But it's equally important that the functionality for the screen part **stay tightly coupled to the screen part** that is rendered.

To solve these needs, Android introduced the idea of **screen fragments**.

On a tablet...

Users will see both fragments on a single screen. And the logic to drive each screen art is in a separate fragment.



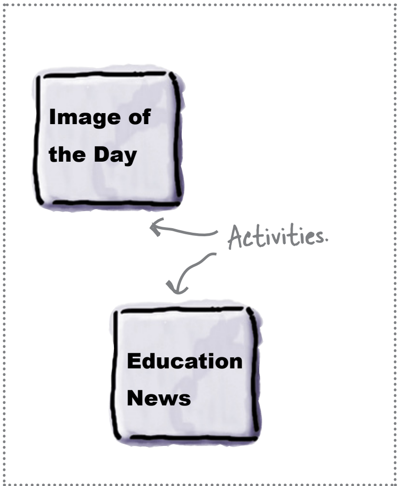
On a phone...

The fragments can be configured to run on separate screens, one for each fragment.

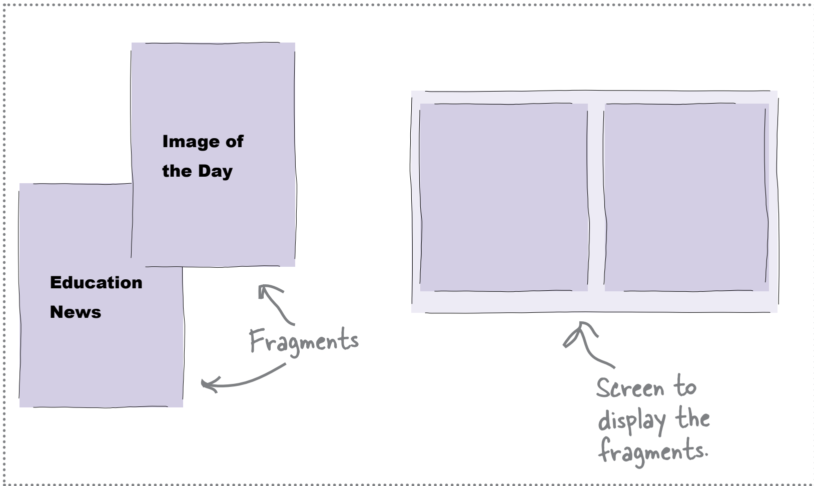
... but the codebase is built on activities

Fragments sound like a great idea, but right now your codebase is built on Activities, not fragments. Without rewriting your app from from scracth, you need to convert those Activities to fragments. **How are you going to do it?**

You have two Activities...



... you want to fragments and a Screen.



Converting your existing app to use fragments

Here are the steps you'll take to convert your existing app to use fragments without rewriting it from scratch.

1. **Convert your existing Activities to fragments.**
   This will allow you to combine them into a single screen.

2. **Create a new Activity that will display the two fragments.**

3. **Add the two fragments to the new Activity.**

4. **Test and bug fix as needed.**

Extend fragment instead of activity

Start by opening the NASA Image of the Day Activity (in `NasaIotd.java`). Change the class declaration from extending `Activity` to extending `Fragment`.

Extending fragment instead of activity.

```
package com.headfirstlabs.ch05.nasa.iotd;

import java.io.IOException;

public class NasaIotd extends Fragment implements IotdHandlerListener {
    private final static String TAG = NasaIotd.class.getName();
    private static final String URL = "http://www.nasa.gov/rss/image_of_the_day.rss";

    private Handler handler;
    private ProgressDialog dialog;
    private Bitmap image;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        handler = new Handler();
        refreshFromFeed();
    }

    private void refreshFromFeed() {
        dialog = ProgressDialog.show(this, "Loading", "Loading the Image of the Day");
        Thread th = new Thread(new Runnable() {
            public void run() {
                IotdHandler iotdHandler = new IotdHandler();
                iotdHandler.setListener(NasaIotd.this);
                try {
                    iotdHandler.processFeed(NasaIotd.this, new URL(URL));
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        });
        th.start();
    }
```

LOTS of errors. (Each line in the margin is an error).



Extending fragment is the right thing to do...

But there are a few other changes you'll need to make in your development environment and the Fragment itself for everything to work seamlessly.

Let's start by updating your development environment...

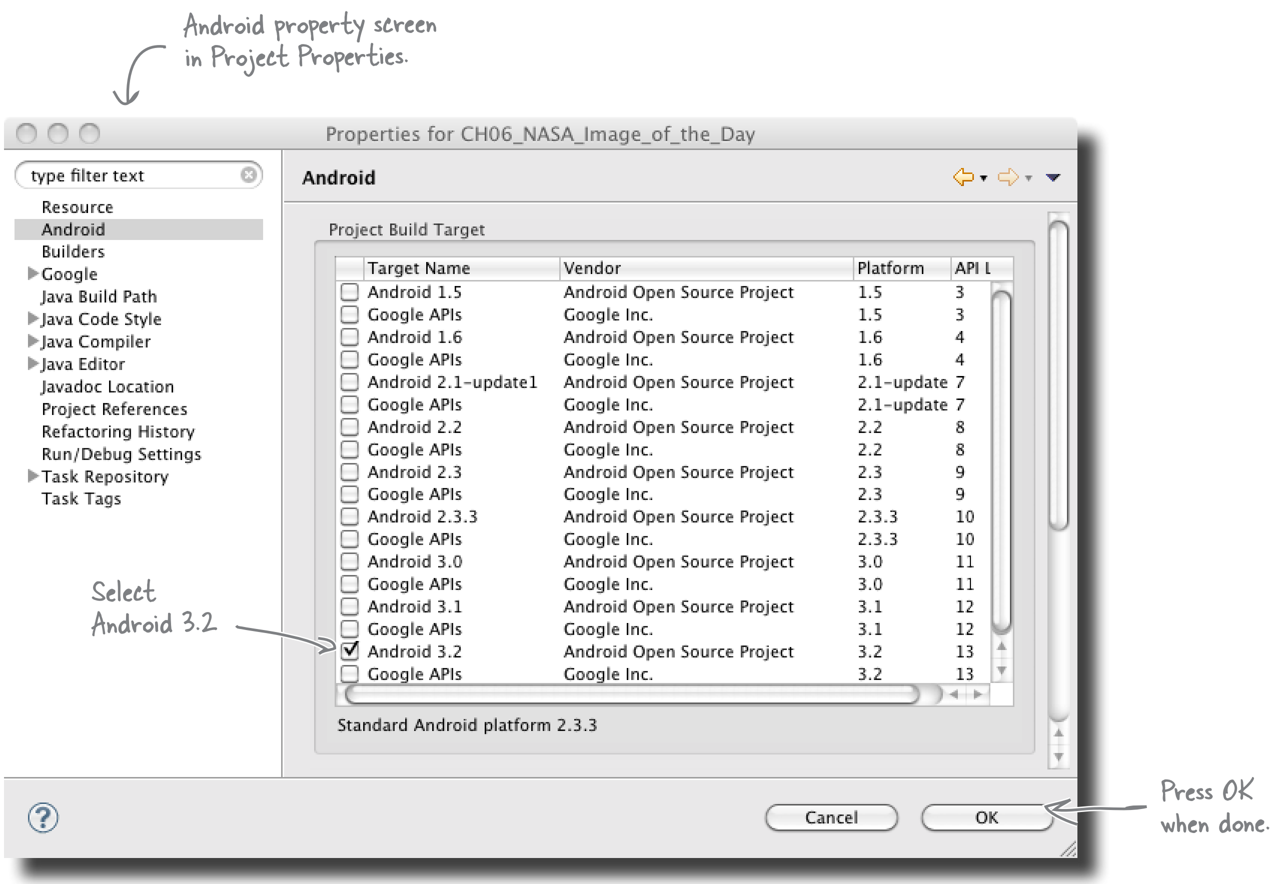
Set the android version for your project

If you hover over the line in the margin of the Eclipse editor, you'll see an error saying that class `Fragment` is not found. Right now, you're probably wondering how the `Fragment` class couldn't be found if you're running the app on a tablet running Android 3.2!

The answer is that even though you're running the app on an Android AVD running Android version3.2, your project is still set to build using version 2.3.3. And `Fragment`'s hadn't been released in Android 2.3.3 so the class `Fragment` couldn't be found.

Do this!

> Update the Android version building this project by going to Project Properties (right click on the project) and select Android. Then select a build target of Android 3.2.



Why do you still get errors?

After you select OK, setting the Android platform version for your project, Eclipse will automatically rebuild your project using the new platform. The `Fragment` class should be found now, but there are still many other errors to contend with.

Fragments aren't activites themselves...

The implementation of the `NasaIotd` relied on several methods inherited by **subclassing** `Activity`. But `Fragment` doesn't extend `Activity`, and now `NasaIotd` extends `Fragment` not `Activity`, so those methods are out of scope.

... but they do have access to their Activity.

Fragments can't be launched by themselves. Instead, your app will still be launched by an `Activity`, and that `Activity` is going to assemble the `Fragments` to display on the screen. But the `Fragment` can get a reference to the `Activity` that added it to a screen using the method `getActivity`.

This code will throw a compiler error

This line is from the `iotdParsed` method and is getting a reference to the Title `TextView` using the `findViewById` method. But that method doesn't exist in `Fragment`.

```
TextView titleView = (TextView)
        findViewById(R.id.imageTitle);
```

findViewById doesn't exist in fragment and will cause a compile error.

This code works

Here is the same line of code modified to work in a `Fragment`. Notice that get`Activity` is called before `findViewById`. This gives a reference to the `Activity` that launched the `Fragment`.

```
TextView titleView = (TextView)
        getActivity().findViewById(R.id.imageTitle);
```

Calling getActivity returns a reference to the Activity that displayed the fragment. Then you can call findViewById

Sharpen your pencil

Below is the `iotdParsed` method that is called when the parser completes parsing and the results are displayed on the screen. Modify the code below to use `getActivity()` to retrieve the `Activity` and make this `Fragment` work correctly.

```
public void iotdParsed(final Bitmap image, final String title,
        final String description, final String date) {

        handler.post(
                new Runnable() {
                        public void run() {
                                TextView titleView = (TextView)
                                        findViewById(R.id.imageTitle);
                                titleView.setText(title);

                                TextView dateView = (TextView)
                                        findViewById(R.id.imageDate);
                                dateView.setText(date);

                                ImageView imageView =  (ImageView)
                                        findViewById(R.id.imageDisplay);
                                imageView.setImageBitmap(image);

                                TextView descriptionView = (TextView)
                                        findViewById(R.id.imageDescription);
                                descriptionView.setText(description);
                        }
                }
        );
}
```

Sharpen your pencil Solution

Below is the `iotdParsed` method that is called when the parser completes parsing and the results are displayed on the screen. You should have modified the code below to use `getActivity()` to retrieve the `Activity` and make this `Fragment` work correctly.

```
public void iotdParsed(final Bitmap image, final String title,
        final String description, final String date) {

        handler.post(
                new Runnable() {
                        public void run() {
                                TextView titleView = (TextView)
                                  getActivity(). findViewById(R.id.imageTitle);
                                titleView.setText(title);

                                TextView dateView = (TextView)
                                  getActivity(). findViewById(R.id.imageDate);
                                dateView.setText(date);

                                ImageView imageView =  (ImageView)
                                  getActivity(). findViewById(R.id.imageDisplay);
                                imageView.setImageBitmap(image);

                                TextView descriptionView = (TextView)
                                  getActivity(). findViewById(R.id.imageDescription);
                                descriptionView.setText(description);
                        }
                }
        );
}
```

All of the findViewById calls need to be preceded by getActivity.

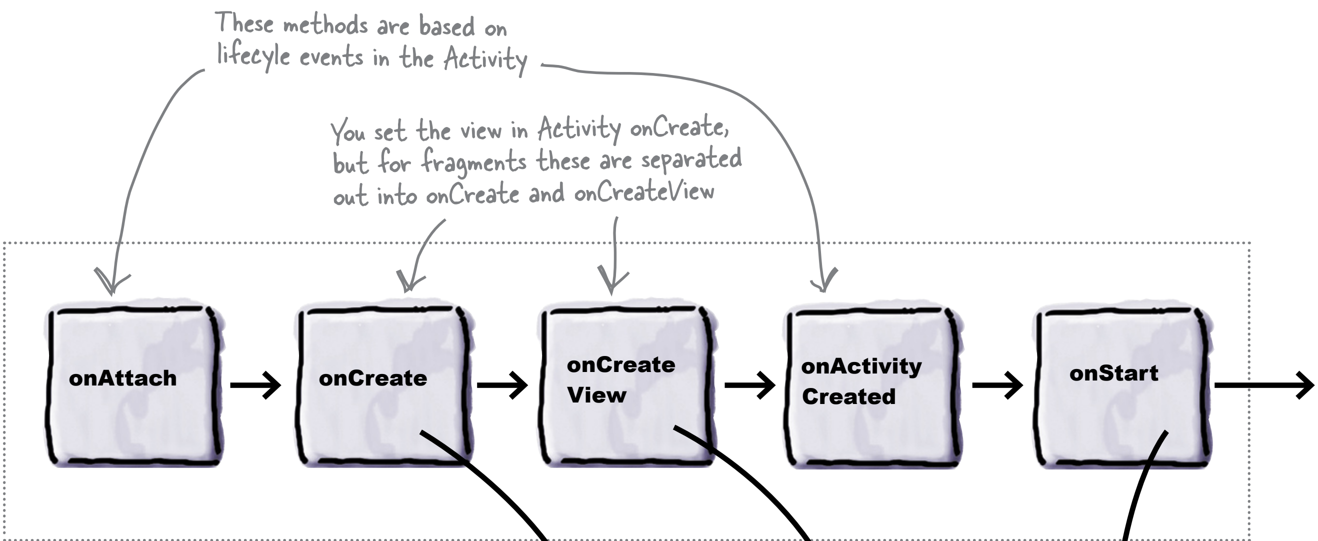
Do this!

Go through the `Activity` and look for other errors that can be fixed by calling `getActivity` before the method call not being found.

Update the lifecycle methods

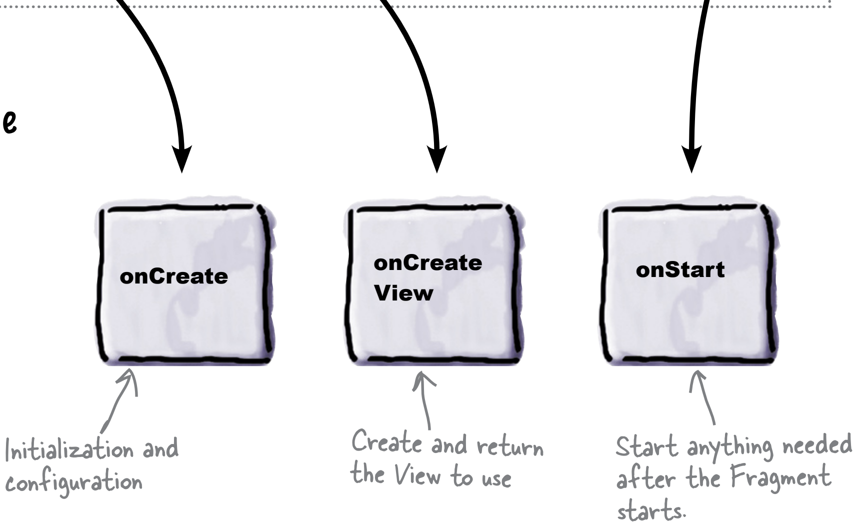
The lifecycle methods of a `Fragment` are also a bit different than `Activity`. One of the major differences are lifecycle methods that allow code to execute when lifecycle methods happen on the associated `Activity`.

Another major change is in `onCreate` in your Activity, you configured the Activity and set the view. With `Fragments`, `onCreate` is separated into two methods, with an additional method added called `onCreateView` which returns a View. This allows the `Activity` to control the view generation and query the `Fragment` for their `Views`.



You'll almost always override these three methods.

Just like `Activity`, there are a number of lifecycle methods you can override from the `Fragment` base class. But you'll almost always override these three.



Verify the lifecycle methods

The only lifecycle method overridden in `NasaIotd` is `onCreate`, which contains all of the initialization and configuration for the `Activity`. When migrating to `Fragments`, this code needs to split up since the lifecycle includes separate creation. attachment, and rendering methods to facilitate combining fragments.

The handler creation and invoking the feed refresh can still happen in `onCreate`.

The handler and refresh can stay in onCreate

```
public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      handler = new Handler();
      refreshFromFeed();
      setContentView(R.layout.activity_main);
}
```

Setting the content view needs to change since Fragments need to return their view in onCreateView rather than setting the content view for the entire Activity.

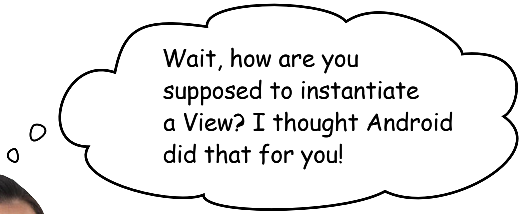
But the view creation has to happen separately. Fragment has a special lifecycle method that returns the View for the `Fragment` called **`onCreateView`**. But rather than returning an `R` constant for the View XML, you return an instantiated `View`.

onCreateView returns an instantiated View.

```
public View onCreateView(LayoutInflater inflater,
      ViewGroup container, Bundle savedInstanceState) {

      //View instantiation goes here ...

}
```

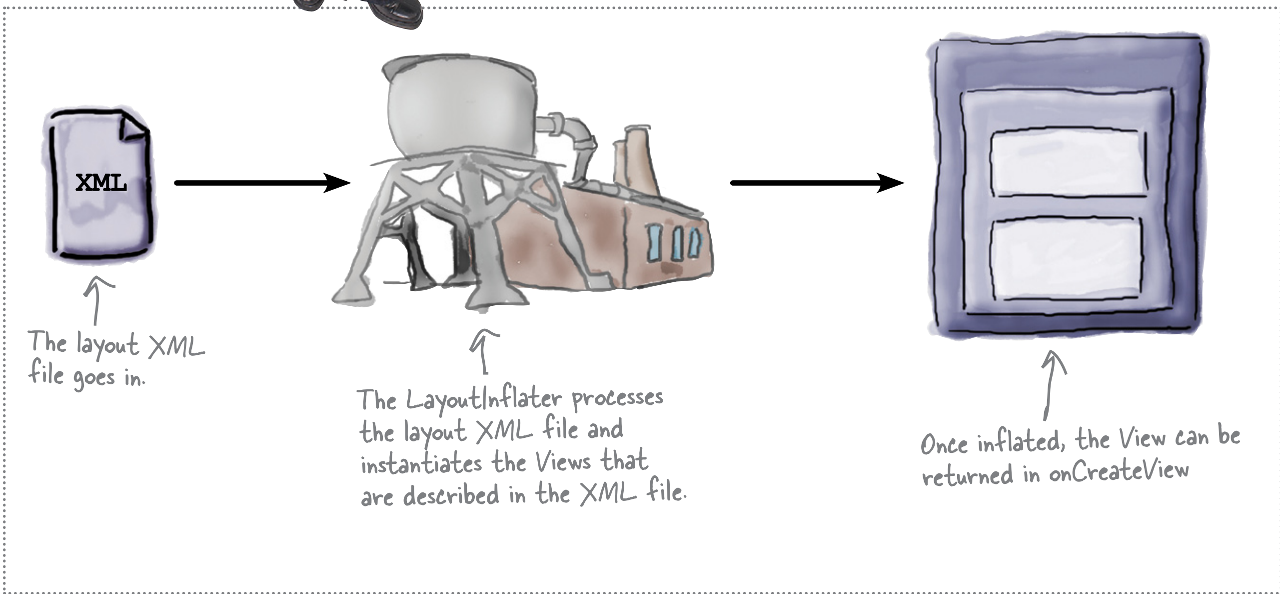




You can instantiate Views yourself, too.

The `setContentView` method is a helper method that takes an `R` file constant and creates a `View`. This works internally by looking up the XML layout defined by that constant, parsing the file, and creating and configuring each `View` specified in the layout.

This creation of real `Views` from layout XML is called *inflation*.



Using LayoutInflater

It sounds like a `LayoutInflater` will do the job, but where do you get the `LayoutInflater`? There are a few parameters passed in to the `onCreateView` method, and one of them is a `LayoutInflater`. And you can use it to inflate your View defined in the `Fragment`'s layout.

A layout inflater is passed in to Fragment's onCreateView.

```
public View onCreateView( LayoutInflater inflater,
      ViewGroup container, Bundle savedInstanceState) {
          You will inflate your
          layout in here
}
```

Inflating the layout

The inflate method on `LayoutInflater` take the `R` constant of the layout you want to inflate as an input parameter. It also takes a root `ViewGroup` that helps the `LayoutInflater` configure the internal layout inflation. The method also takes a `boolean` parameter indicating whether or not to attach the layout to the `ViewGroup` being passed in.

The inflate method from LayoutINflater

```
public void inflate (
      int layoutId,
      ViewGroup root,
      boolean attachToRoot
);
```

- `int layoutId` — The R constant for the layout you want to inflate.
- `ViewGroup root` — The ViewGroup root to configure layout inflation.
- `boolean attachToRoot` — Indicating whether or not to attach the inflated view to the ViewGroup. This is going to be false for fragments.



onCreateView Magnets

Below are empty methods for `onCreate`, `onCreateView`, and `onStart`. Complete the methods with the magnets below paying close attention to which code belongs in each method. Not all of the magnets will be used, so you will have some left over.

```
public void onCreate(Bundle bundle) {

}
```

Initialization and configuration code in here.

```
public View onCreateView(LayoutInflater inflater,
              ViewGroup container, Bundle savedInstanceState) {

}
```

Inflate and return the view for the fragment

```
public void onStart() {

}
```

Call the refresh method in here to be sure the view was created

`super.onStart();`

`handler = new Handler();`

`setContentView(`

`);`

`refreshFromFeed();`

`return`

`R.layout.activity_main`

`false`

`container,`

`inflater.inflate(`



onCreateView Magnets Solution

Below are empty methods for `onCreate`, `onCreateView`, and `onStart`. You should have completed the methods with the magnets below paying close attention to which code belongs in each method. You should have extra magnets left over.

```
public void onCreate(Bundle bundle) {
    super.onCreate(bundle);
    handler = new Handler();
}
```

The handler can still be created in onCreate.

```
public View onCreateView(LayoutInflater inflater,
              ViewGroup container, Bundle savedInstanceState) {
    return inflater.inflate(
        R.layout.activity_main container, false );
}
```

Inflate the layout using the layout inflater passed in, and inflating the activity_main layout.

```
public void onStart() {
    super.onStart();
    refreshFromFeed();
}
```

Refresh the feed in on start, this way, onCreateView will have already been called and the views will have been created.

```
setContentView(
```

There is no need to call setContentView since onCreateView returns the view.

Convert NasaEdNews to a fragment

You're done converting the NasaIotd to a Fragment, but now you have to do the same updating to `NasaEdNews`. They both need to be `Fragments` so you can add them to the screen.

Do this!

Update your version of `NasaEdNews` to be a `Fragment` according to these changes.

Extend fragment instead of Activity.

```
public class NasaEdNews extends Fragment implements EdNewsHandlerListener {
    private static final String URL = "http://www.nasa.gov/rss/educationnews.rss";
    private Handler handler;
    private EdNewsAdapter listAdapter;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        handler = new Handler();
        setContentView(R.layout.ed_news);
    }

    public View onCreateView(LayoutInflater inflater,
        ViewGroup container, Bundle savedInstanceState) {
        return inflater.inflate(R.layout.ed_news, container, false);
    }

    public void onStart() {
        super.onStart();

        listAdapter = new EdNewsAdapter();
        ListView listView = (ListView)
        getActivity().findViewById(R.id.ed_news_list);
        listView.setAdapter(listAdapter);

        refreshFromFeed();
    }
```

Don't call setContentView (the `setContentView(R.layout.ed_news);` line is struck through)

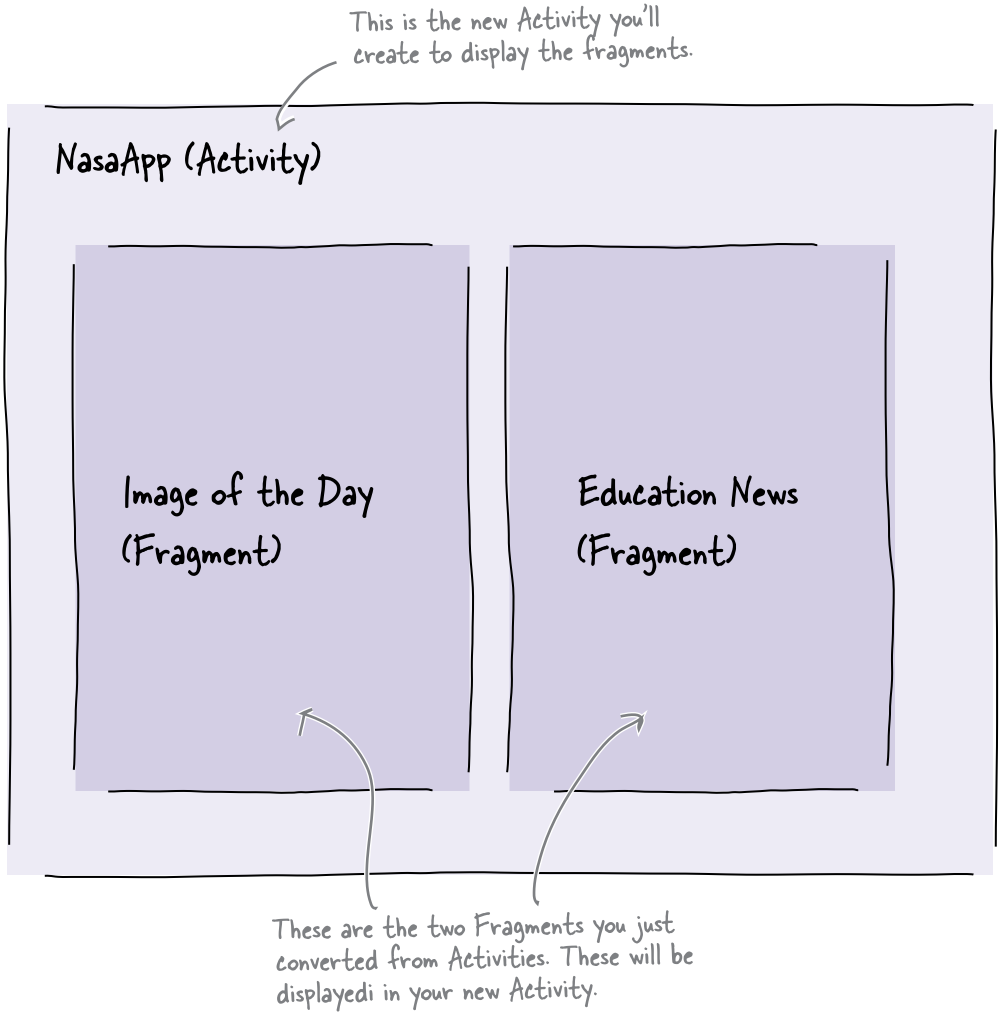
Inflate the layout in onCreateView.

Get a reference to the Activity, then call findViewById

Make the surrounding activity

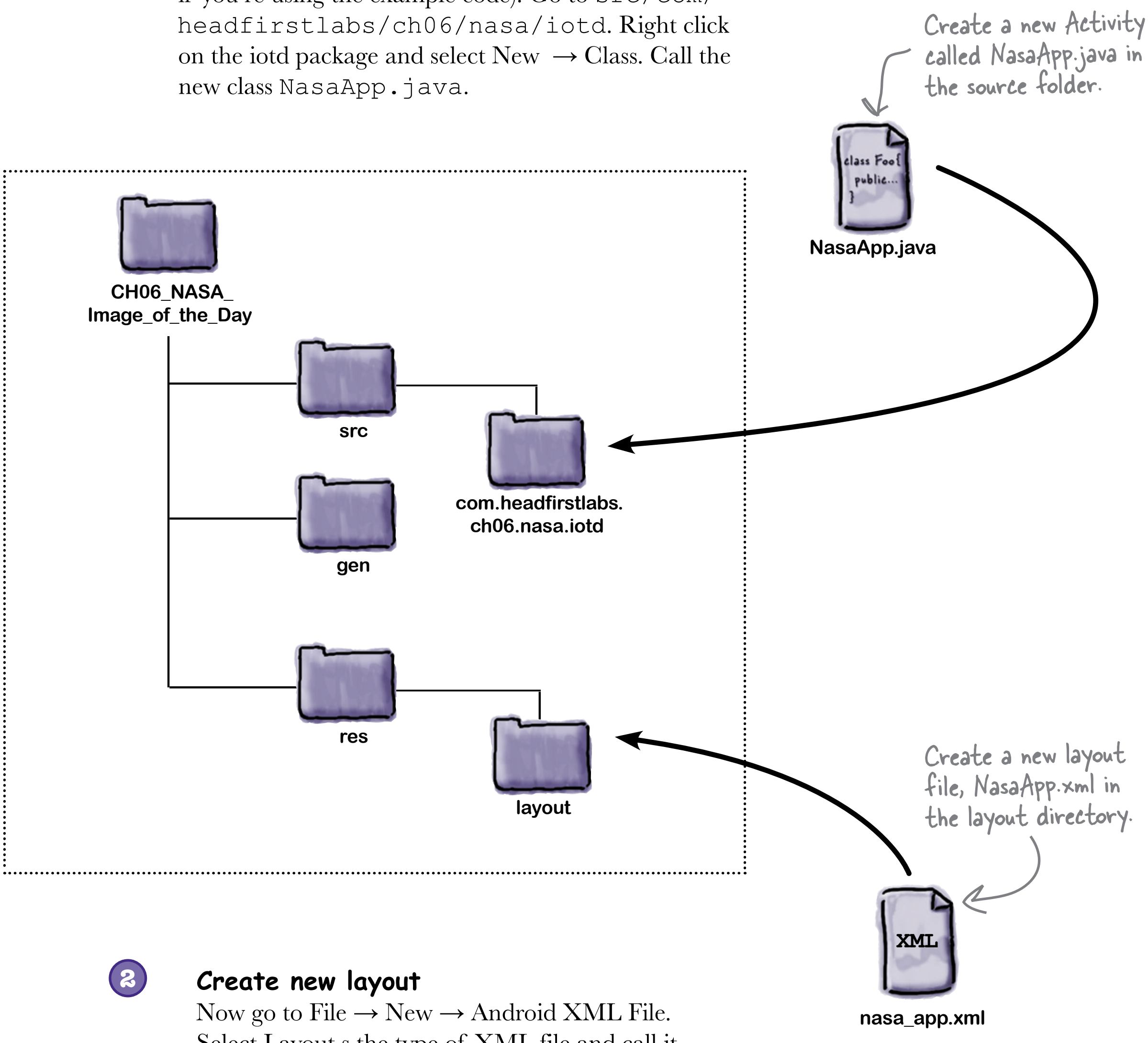
Now you've converted both Activities to Fragments, but you can't launch a Fragment on it's own. You can combine the Fragments in an Activity and display the Activity... but you don't have an Activity in your app.

Now you'll make a new Activity, and render the Image of the Day and Education News Fragments in that new Activity.



1 **Create a new Activity java class**

In the **Eclipse Package Explorer**, navigate to the project source (`CH06_NASA_Image_of_the_Day` if you're using the example code). Go to `src/com/headfirstlabs/ch06/nasa/iotd`. Right click on the iotd package and select New → Class. Call the new class `NasaApp.java`.



2 **Create new layout**

Now go to File → New → Android XML File. Select Layout s the type of XML file and call it `nasa_app.xml`.

Built out a basic Activity

You just created a class and a layout for the Activity, but they are both empty. Start by building out `NasaApp.java` to extend Activity, and create an `onCreate` method to render the layout.

```
public class NasaAppActivity extends Activity {
      protected void onCreate(Bundle savedInstanceState) {
             super.onCreate(savedInstanceState);
             setContentView(R.layout.nasa_app);
      }
}
```



NasaApp.java



NasaApp is an Activity, not a Fragment.

You just converted both `NasaIotd.java` and `NasaEdNews.java` to be `Fragments` instead of `Activities`, and `Fragments` should use `onCreateView` which returns a `View` instead of setting the **Content View**. But `NasaApp.java` is an Activity, and it should set the **Content View**.

Update the manifest

The **AndroidManifest.xml** contains metadata about how your app is configured, and how to run and install it. You first modified the manifest file in **Chapter 3** to add permissions to access the network. The manifest file also includes a reference to the `Activity` to launch. And since you're changing the `Activity` to launch (from `NasaIotd` to `NasaApp`) you need to update the manifest.

```
<application android:icon="@drawable/icon" android:label="@string/app_name" >
    <activity android:name=".NasaApp"
        android:label="@string/app_name" >
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
</application>
```

Update the android:name attribute in the Activity to .NasaApp from .NasaIotd since NasaApp is the new Activity.



AndroidManifest.xml

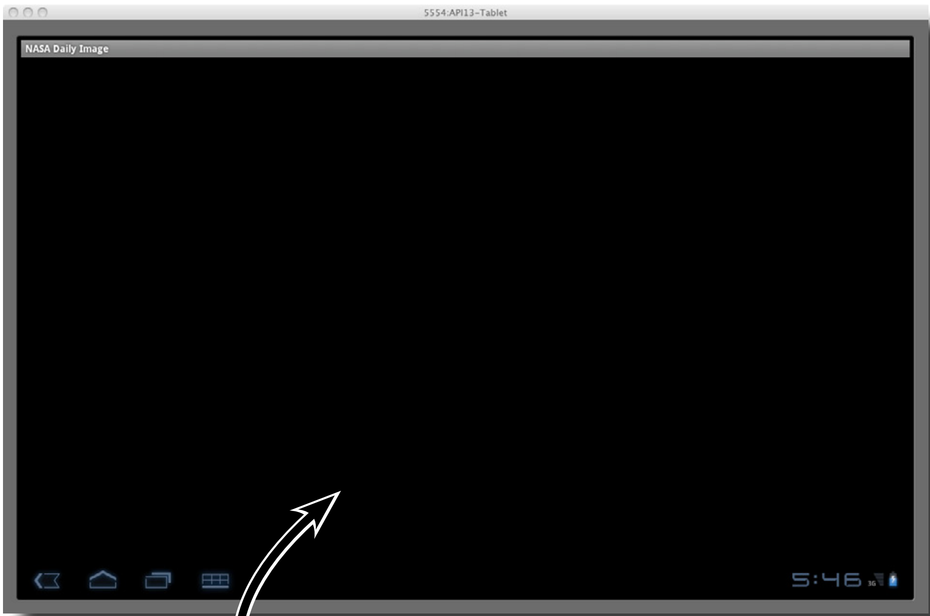


Open the `AndroidManifest.xml` file in the project root. Click on the tab to the right labeled `AndroidManifest.xml` to edit the XML directly. Update the android:name attribute in the `Activity` to point to the new `Activity` you just created.

Test Drive

Run the app now to verify everything is starts and renders the Activity

The app runs, but the screen is empty. This isn't surprising since you are displaying a layot that has no Views.



The layout you just created in NasaApp.xml does not contain any Views yet.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="horizontal"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

</LinearLayout>
```



nasa_app.xml



It's no surprise the layout is empty...

You created the new Activity and layout, but you haven't populated it yet. The fragments need to be displayed on the screen too.

Now that you have the completed Fragments and an Activity to display them, ***let's see how to display the Fragments on screen.***

Add the fragments to your layout

Fragments can be added to a screen in the XML layout. There is a special **`<fragment>`** element added to XML layouts after Fragments were introduced.

It's a good idea to add an android:id attribute for your fragment. This will allow you to retrieve and configure the fragment from your activity later on.

The fully qualified class name of your fragment goes here

```
<fragment android:name="_______________"
    android:id="_______________"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

You can use regular layout attributes on a fragment just like any other View.

there are no Dumb Questions

Q: You're defining a fragment in the layout, but assigning View layout attributes. Can you do that?

A: Yes. You're defining the fragment attribute and referencing the fragment class. But the view is rendered to the screen, and view attribute control how the view is laid out.

Q: Do I have to do any other configuration to make the fragment load?

A: No, defining the fragment in the layout renders the view and instantiates the fragment class.

Q: Do I have to start the fragment or call any of the other lifecycle methods?

A: Nope! That's all done for you automatically.

Q: Do I have to declare it in the layout? What if I want to programatically decide which Fragment to add?

A: You can add fragments programatically in addition to declaring them in the layout. Check the online docs for more information.

Exercise

Below is the empty layout for `nasa_app.xml`. Add both the `NasaIotd` and the `NasaEdNews` `Fragments` to this layout. Add them to the current horizontal `LinearLayout` below. Make sure to give the `Fragments` `android:id` attributes as you would for other Views.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
       android:orientation="horizontal"
       android:layout_width="match_parent"
       android:layout_height="match_parent">




</LinearLayout>
```

Exercise Solution

Below is the empty layout for `nasa_app.xml`. You should have add both the `NasaIotd` and the `NasaEdNews Fragments` to this layout. You should have added them to the current horizontal `LinearLayout` below. You also should have given the `Fragments` `android:id` attributes as you would for other Views.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="horizontal"
        android:layout_width="match_parent"
        android:layout_height="match_parent">

    <fragment android:name="com.headfirstlabs.ch06.nasa.iotd.NasaIotd"
        android:id="@+id/fragment_iotd"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="1" />

    <fragment android:name="com.headfirstlabs.ch06.nasa.iotd.NasaEdNews"
        android:id="@+id/fragment_ed_news"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="1" />

</LinearLayout>
```

The image of the day fragment

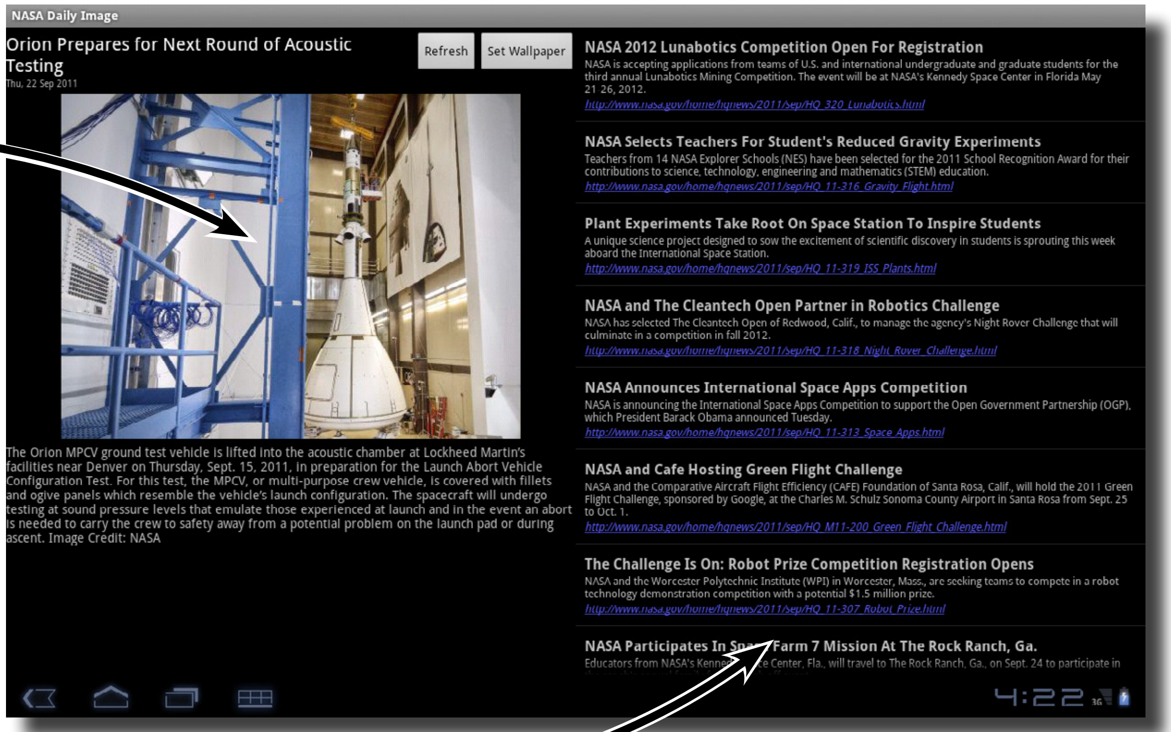
Fully qualified class names of the fragments

The education news fragment



Test Drive

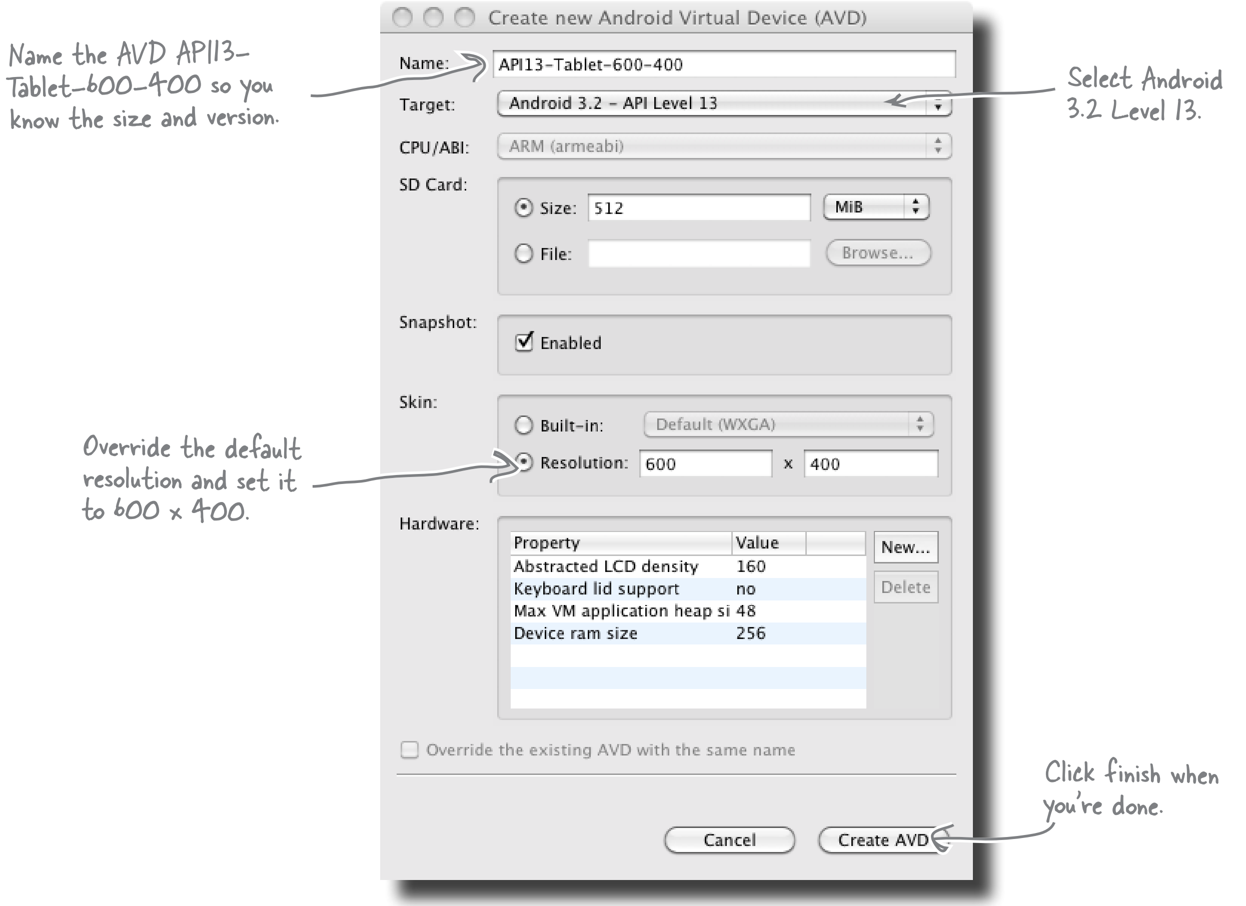
Run the app again. At this point, you should see both of the fragments displaying on the screen.



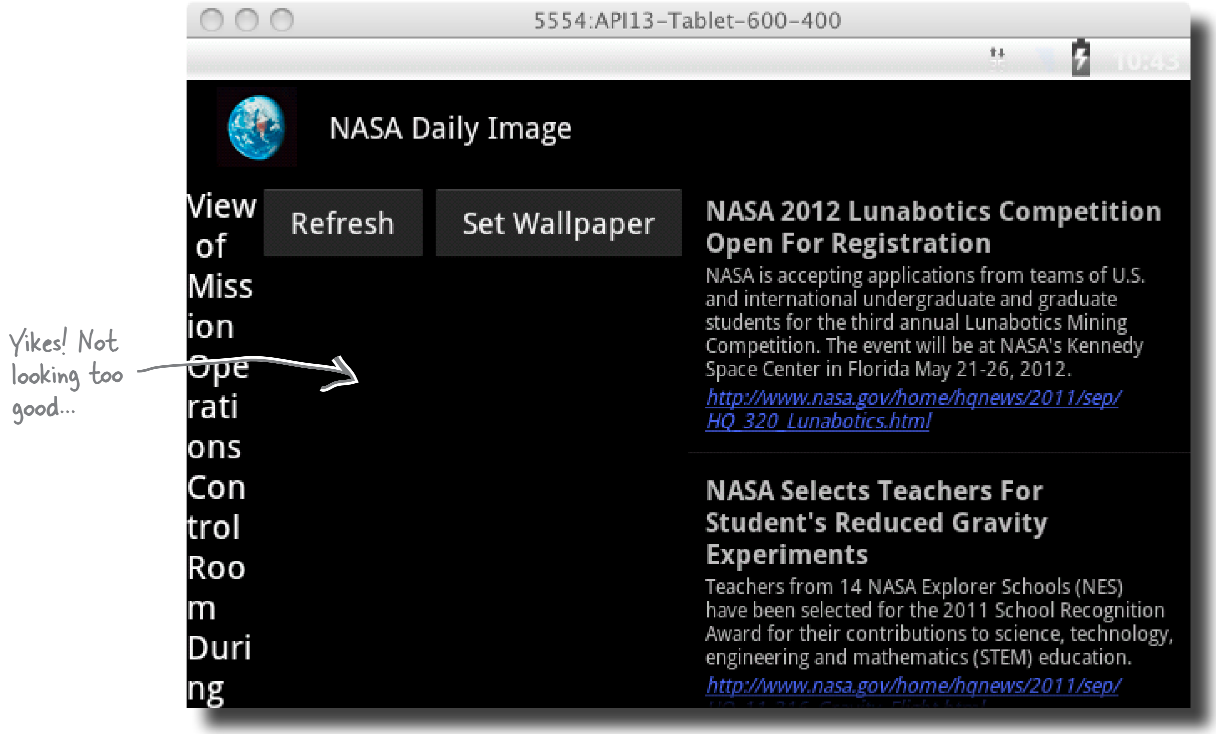
This screen is just a big horizontal LinearLayout with two large Views... the two fragments!

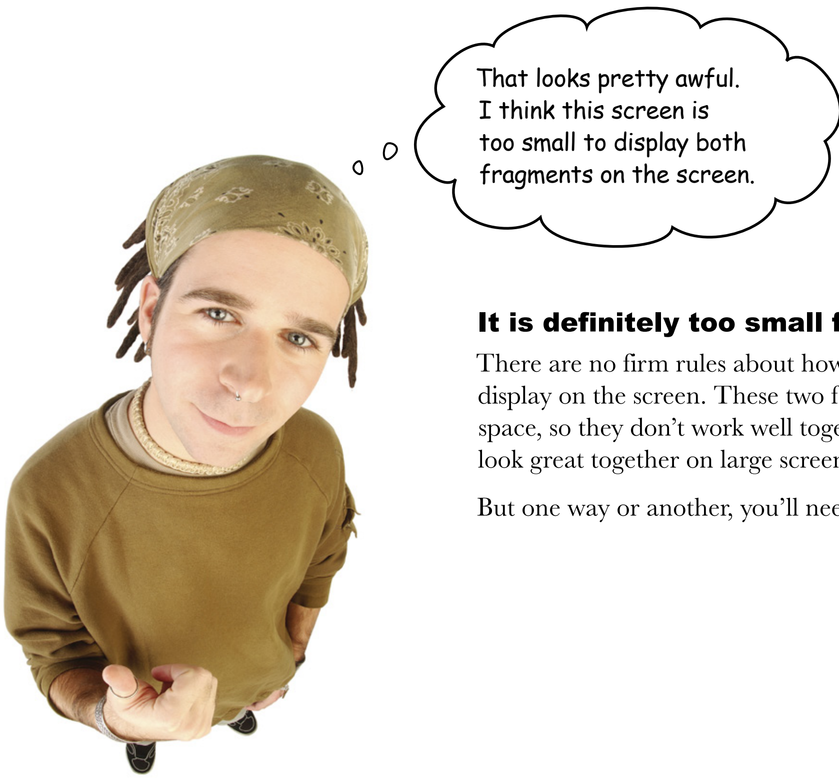
Test it on a small screen

Right now you're testing the app on the default tablet size for the emulator, **WXGA**, which is a sizable **1280x800**. But not all of your users' devices are going to be that big, even tablets. Let's make a small Android Version 3.2 AVD to see how the app looks.



Launch the emulator and run the app.





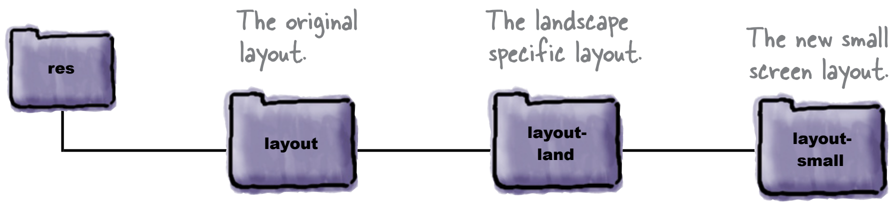
It is definitely too small for these fragments.

There are no firm rules about how many fragments you can display on the screen. These two fragments take up a lot of space, so they don't work well together on small screens. They look great together on large screens though.

But one way or another, you'll need to fix this on small screens...

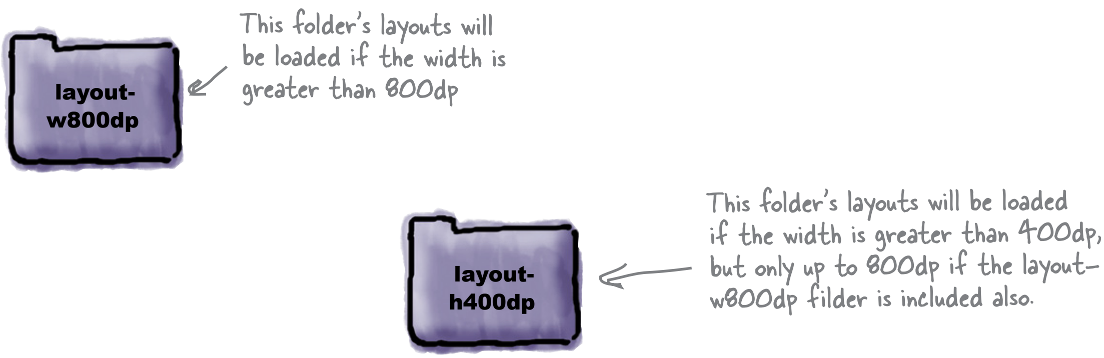
New Screen Groups Up Close

In ***Chapter 5***, you build optimized layouts for small screen devices and landscape. Using the same **R** constant, layouts are *dynamically loaded* based on their screen size category (**small**, **med**, **large**, and a recent addition, **x-large**). These are your layout folders from ***Chapter 5***.



Android 3.0 introduced the idea of minimium screen widths and heights to determine the dynamic layout loading. So instead of declaring **small**, **med**, or **large** screen widths, you can declare screen widths in **Density Independent Pixels** (**DPs**).

The name of the folder determines in the screen size the layout applied for. And just like the screen group following the layout in the folder name, so does the screen size. The difference is that the width or height is specficed with a **w** or **h**, followed by the dimension in DPs. ***Screens larger then the specified width or height load the layouts in the folder.***



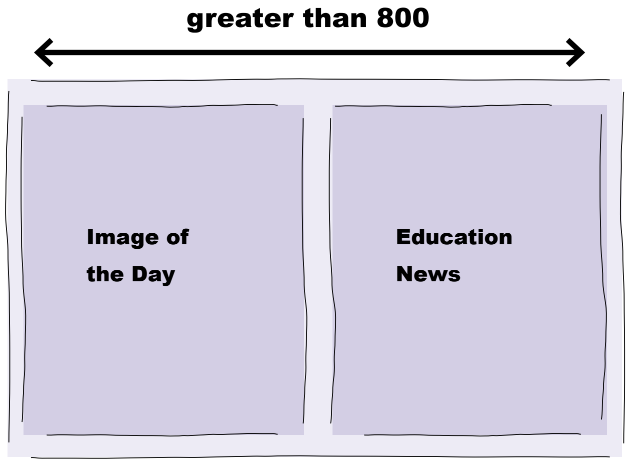
For newer versions of Android, either the old style screen groupings or the new minimum screen size approach will work. However, older versions of Android require the older style screen grouping approach. The new and old screen grouping approaches can work together, and you'll need to do that if you plan on supporting older versoins and new versions of Android in the same app.

Use two optimized layouts

Since both of the fragments won't really fit on the small **600x400** screen, let's make special layouts for large and small screens according to the minimum screen width and height optimized layouts in 3.0.

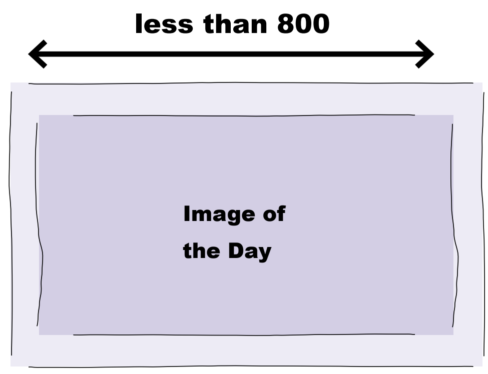
One large layout

If the screen is large enough, display both of the fragments side by side. Depending on your target devices and application content, this size might vary. For the Nasa App, let's define the minimum size as **800** pixels for side by side fragments. This will be a new layout specifying screen sozes 800 pixels wide or above.



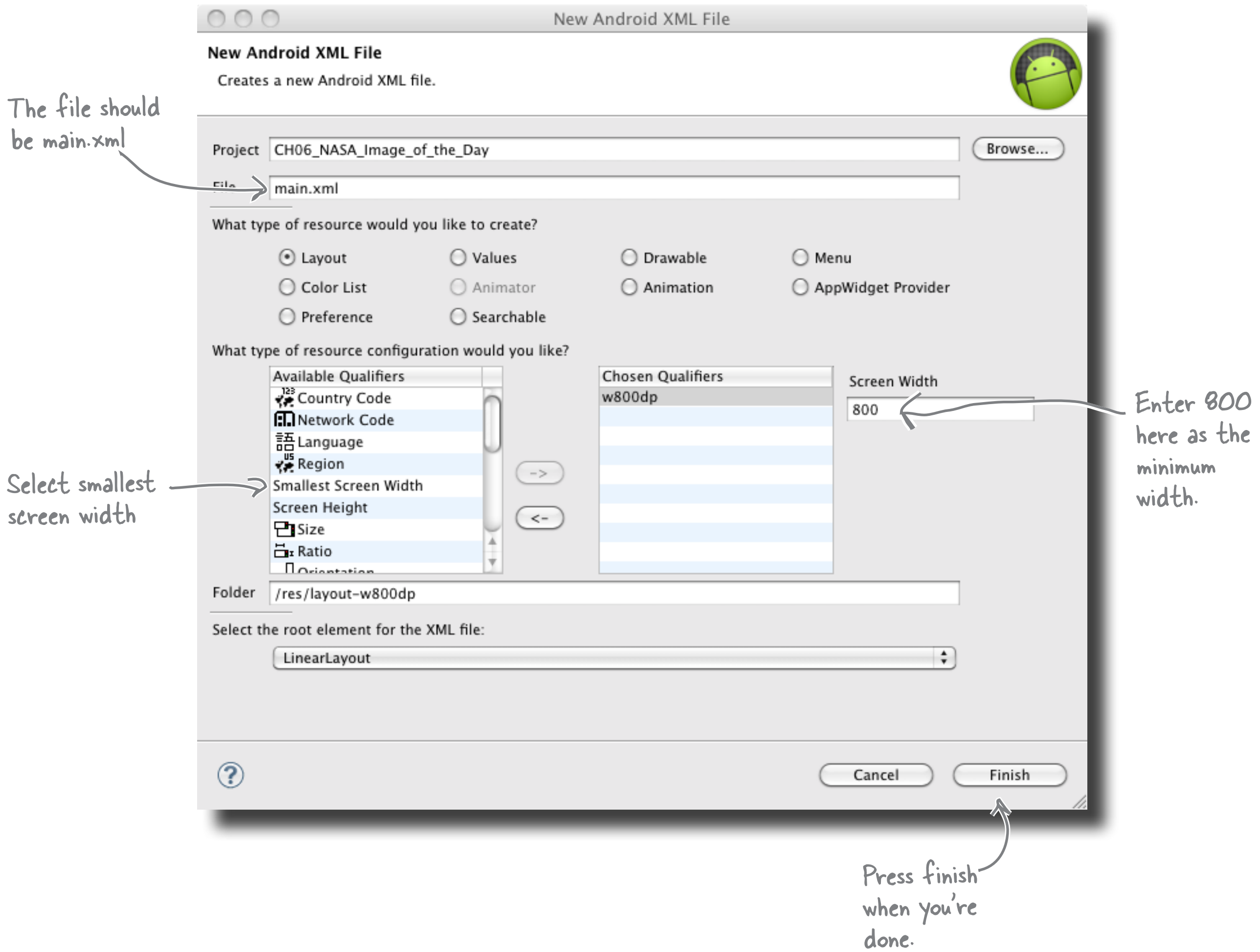
One small layout

If the app is less that **800** pixels, just display the **Image of the Day** fragment and ***not*** the **Education News**. This will be the layout in /`res/layout/main.xml`.



Create the large screen layout

Just like the landscape mode, newer versions of the Android Eclipse Plugin allow you to configure the minimum layout width directly from the new Android XML File wizard. Launch the wizard now and create the large screen optimized XML layout.





Screen Support Exposed

This week's interview:
How do you keep it all straight?

Head First: Thanks for taking time out of your busy schedule of laying out screens and determining which layouts to use to come and talk to us.

Screen Support: A pleasure, as always.

Head First: You know, I thought the small, normal, and large screen sizes which seemed a little hard to keep track of. Then I learned about screen pixel density and seemed like a LOT to keep track of.

Screen Support: Ah yes, the good old days.

Head First: The good old days?

Screen Support: Yes. Back then I just had one system of screen sizes to keep track of. Now I have to keep track of all of that, plus the new system of defining widths directly in the folder name.

Head First: I honestly don't know how you do it.

Screen Support: Oh it's not that bad. I just have an algorithm I follow to figure out which resource to use. It's not like I'm making random decisions myself or anything.

Head First: But don't developers get frustrated trying to nail down the different resources?

Screen Support: Some do. But they get used to my algorithm and then they know what layouts to build for what screen sizes. And they know which to override to make some device work the way they want.

Head First: I'm still shocked that this doesn't confuse you, all of these different layouts in the same app! It would drive me nuts!

Screen Support: If you want to really go nuts, check out all of the other overrides you can do for each layout in addition to size and pixel density.

Head First: You're kidding, there's even more? I had a hard enough time keeping up with this already!

Screen Support: Sure! You can also override layouts by input type. Say for example you have an app with an on screen numeric keyboard for touch screens. You can customize the layout for 12-key devices to remove the keyboard since they already have hardware buttons.

Head First: OK, this is just getting out of hand.

Screen Support: And I'm not even done! You can also customize your layouts by locale. Say for example, you're working with a language that reads right to left instead of left to right. You can add customized layouts that reverse parts of the screen for those languages.

Head First: Enough already! You lost me with the 12 key devices!

Screen Support: Like I said though, it's a piece of cake.

Head First: Of course, its the algorithm right?

Screen Support: Sure is! It's all in the algorithm. As long as we both follow the same rules, there will be no nasty surprises!

Head First: I'm going to take your word for it.

Screen Support: Suit yourself! Just remember, I'm here when you need me.



Long Exercise

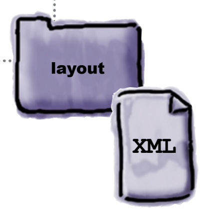
Below are the layouts for the `main.xml` in both the the `res/layout` folder and the `res/layout-w800dp` folder. Modify the `main.xml` in the layout folder to display just the `NasaIotd` fragment for small screen devices. Also modify the currently empty layout in `layout-w800dp/main.xml` to show both fragments.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
       android:orientation="horizontal"
       android:layout_width="match_parent"
       android:layout_height="match_parent">

       <fragment android:name="com.headfirstlabs.ch06.nasa.iotd.NasaIotd"
              android:id="@+id/fragment_iotd"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:layout_weight="1" />

       <fragment android:name="com.headfirstlabs.ch06.nasa.iotd.NasaEdNews"
              android:id="@+id/fragment_ed_news"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:layout_weight="1" />

</LinearLayout>
```



nasa_app.xml

```
<?xml version=”1.0” encoding=”utf-8”?>
<LinearLayout xmlns:android=”http://schemas.android.com/apk/res/android”
       android:orientation=”horizontal”
       android:layout_width=”match_parent”
       android:layout_height=”match_parent”>

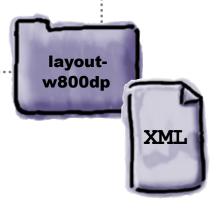








</LinearLayout>
```



nasa_app.xml

Long Exercise Solution

Below are the layouts for the `main.xml` in both the the `res/layout` folder and the `res/layout-w800dp` folder. Modify the `main.xml` in the layout folder to display just the `NasaIotd` fragment for small screen devices. Also modify the currently empty layout in `layout-w800dp/main.xml` to show both fragments.

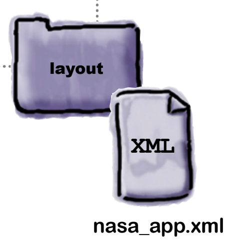
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
       android:orientation="horizontal"
       android:layout_width="match_parent"
       android:layout_height="match_parent">

       <fragment android:name="com.headfirstlabs.ch06.nasa.iotd.NasaIotd"
              android:id="@+id/fragment_iotd"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:layout_weight="1" />

       <fragment android:name="com.headfirstlabs.ch06.nasa.iotd.NasaEdNews"
              android:id="@+id/fragment_ed_news"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:layout_weight="1" />

</LinearLayout>
```

The small screen layout in layout/main.xml should only contain the NasaIotd fragment. So remove the education news fragment.



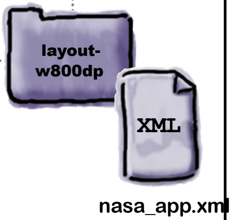
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="horizontal"
        android:layout_width="match_parent"
        android:layout_height="match_parent">

    <fragment android:name="com.headfirstlabs.ch06.nasa.iotd.NasaIotd"
            android:id="@+id/fragment_iotd"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="1" />

    <fragment android:name="com.headfirstlabs.ch06.nasa.iotd.NasaEdNews"
            android:id="@+id/fragment_ed_news"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="1" />

</LinearLayout>
```

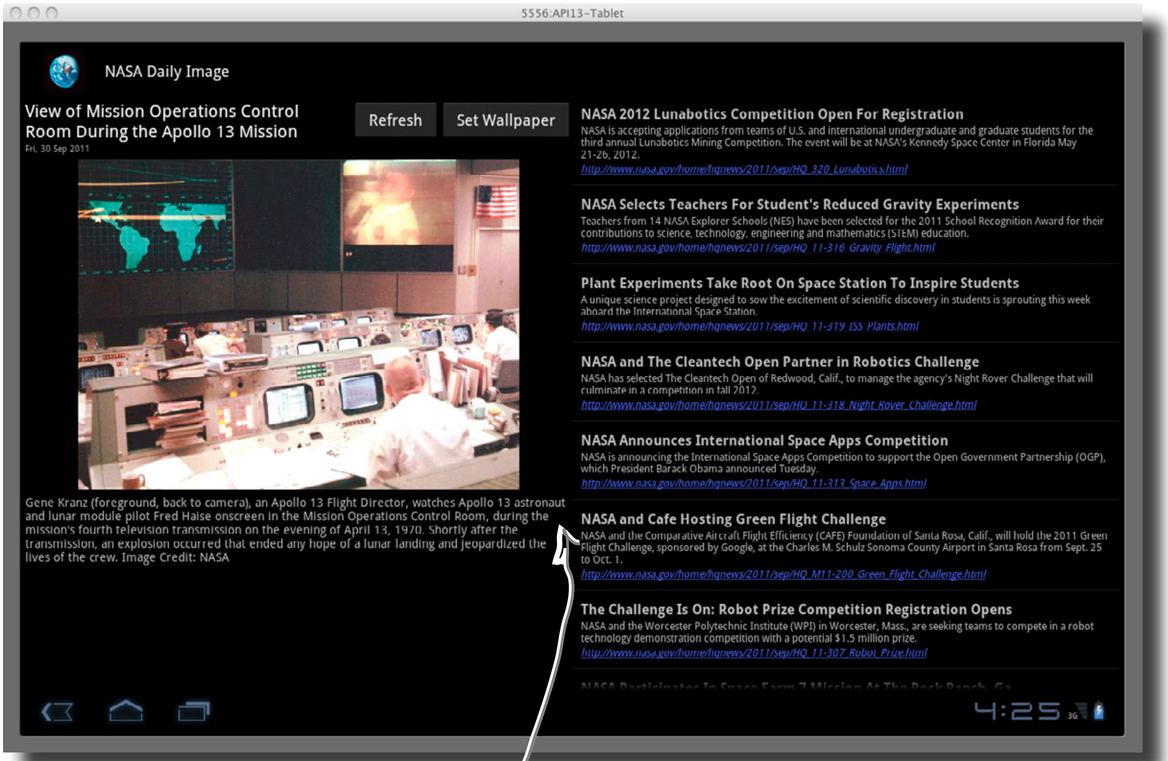
The entire contents of layout/main.xml move to the large screeb layout.



nasa_app.xml

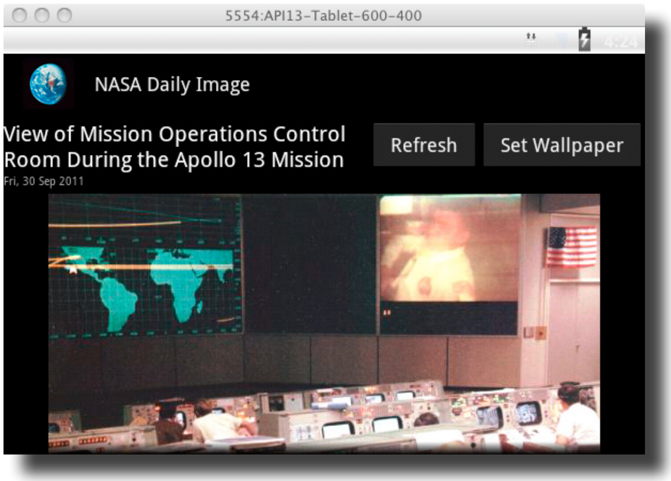
Test Drive

Now that you have optimized layouts for small screen and large (over 800dp width) devices, run the app again and make sure it works on both devices. Use the AVD selection dialog in Eclipse to run the app on both AVDs if you are running them at the same time.



The large screen format still looks good.

And the small screen format looks great too.





That looks great! Add more content only where it works. Perfect!

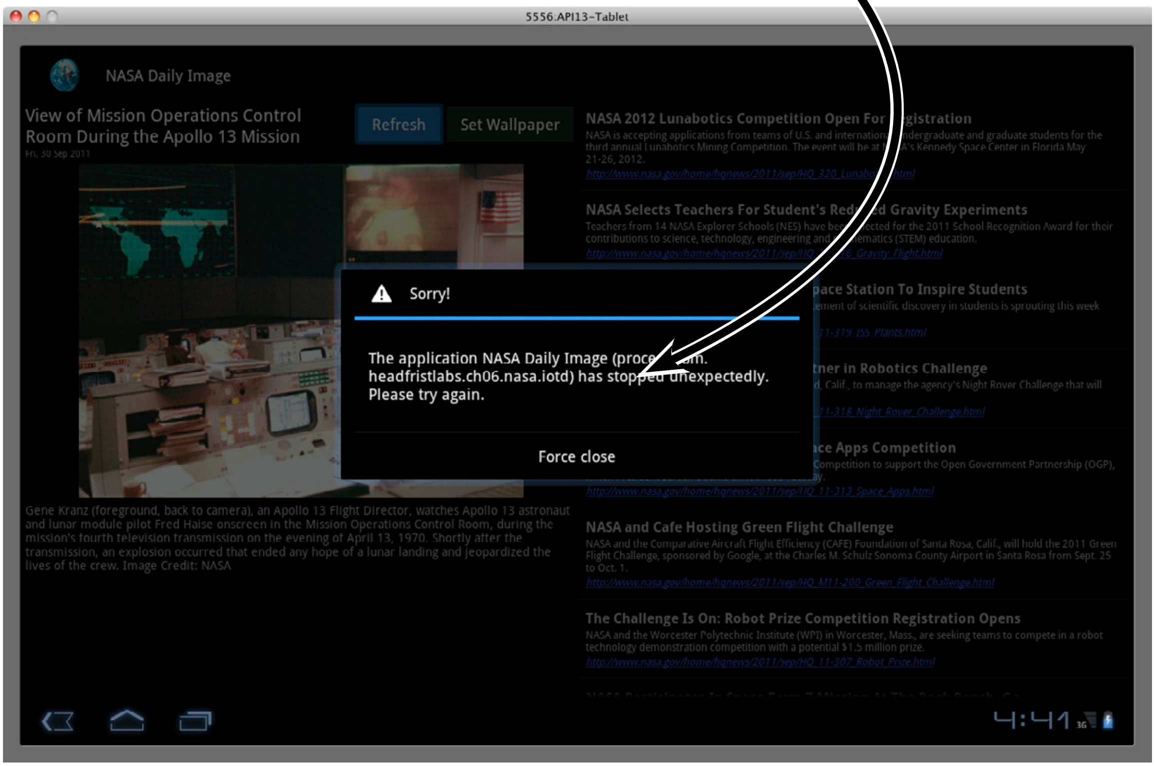
Fragments made is all possible

This is a perfect example of customizing your app to render more or less content based on screen size. And with fragments, it was easy to just add or remove content (and the functionality to support the content like the feed refreshing) just from your layouts without having to move a lot of code around.

Test the app functionality

Speaking of functionality, this is a great time to test the app and make sure everything works. You already know the feed is refreshing correctly, but what about scrolling and the on screen buttons?

Ouch! The app crashed!



Why is the app crashing?

The feeds are loading correctly, and scrolling works. But when you press the butons, the app is crashing. Here is the output.

```
java.lang.IllegalStateException:
      Could not find a method
      onRefreshButtonClicked(View) in the activity
      class com.headfirstlabs.ch06.nasa.iotd.NasaApp
      for onClick handler on view class android.widget.Button
      with id 'refreshButton'
```

BRAIN POWER

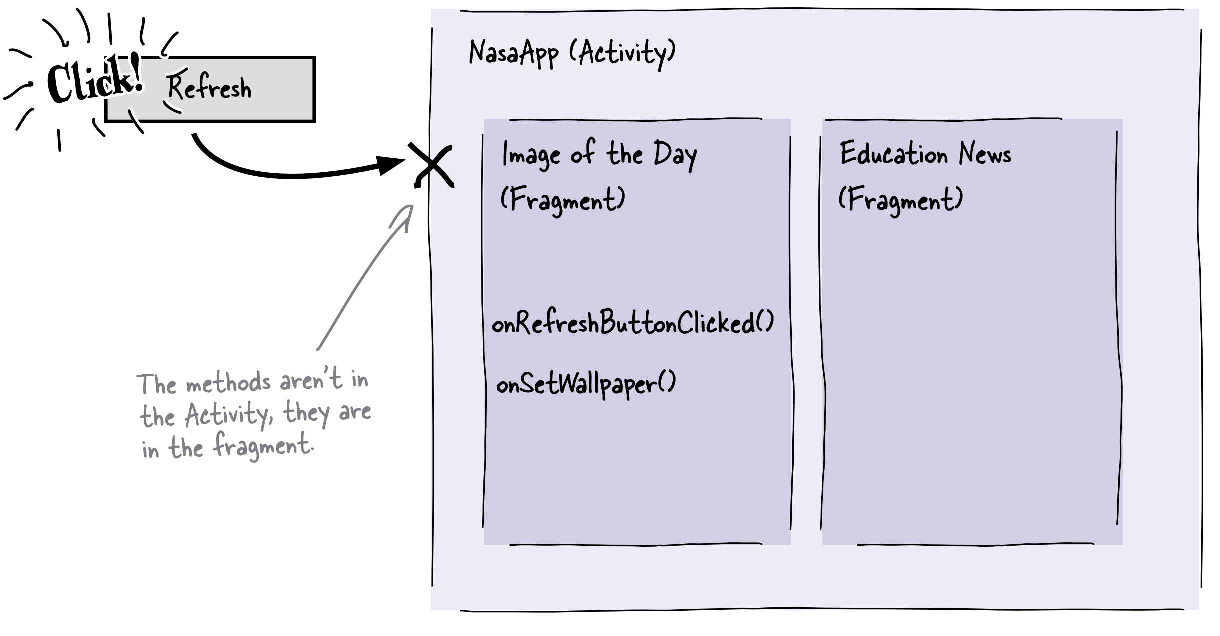
Can you figure out what the error is referring to? Why might this error be occurring? How would you fix it?

Add onClick methods to the activity

To make the buttons work, you added `android:onClick` attributes to the buttons and corresponding methods in the `NasaIotd` Activity. There's just one big problem...

NasaIotd isn't the Activity anymore, it's a Fragment.

`NasaApp` is the Activity now. So even though you have the corresponding onClick methods in `NasaIotd`, the Android action code is looking for the `android:onClick` methods in `NasaApp`.



Make the buttons work

Both of these methods are already implemented in `NasaIotd`, they just aren't receiving the event since they are a *Fragment* not an *Activity*. So all you really need to do is pass the event to the *Fragment.*

You can pass the event to the `Fragment`, but first you need to get a reference to the Fragment from the `Activity` so you can call the `onClick` methods in the `Fragment`.

That is where the **`FragmentManager`** comes in. The `FragmentManager` allows you to retrieve references to `Fragments`. The following code implements both of the onClick methods, retrieves the `FragmentManager` and calls the underlying method in the Fragment. `Fragment`

Do this!

Add these two methods to the `NasaApp Activity`. These receive the expected `onClick` calls and pass them along to the underlying `Fragment`.

Get the FragmentManager.

```
public void onRefreshButtonClicked(View view) {
    FragmentManager fragmentManager = getFragmentManager();
    NasaIotd nasaIotdFragment = (NasaIotd)
            fragmentManager.findFragmentById(R.id.fragment_nasa_iotd);
    nasaIotdFragment.onRefreshButtonClicked(view);
}
```

Pass the call through to the fragment.

Find the fragment using findFragmentById.

```
public void onSetWallpaper(View view) {
    FragmentManager fragmentManager = getFragmentManager();
    NasaIotd nasaIotdFragment = (NasaIotd)
            fragmentManager.findFragmentById(R.id.fragment_nasa_iotd);
    nasaIotdFragment.onSetWallpaper(view);
}
```

Implement the same process for onSetWallpaper

NasaApp.java

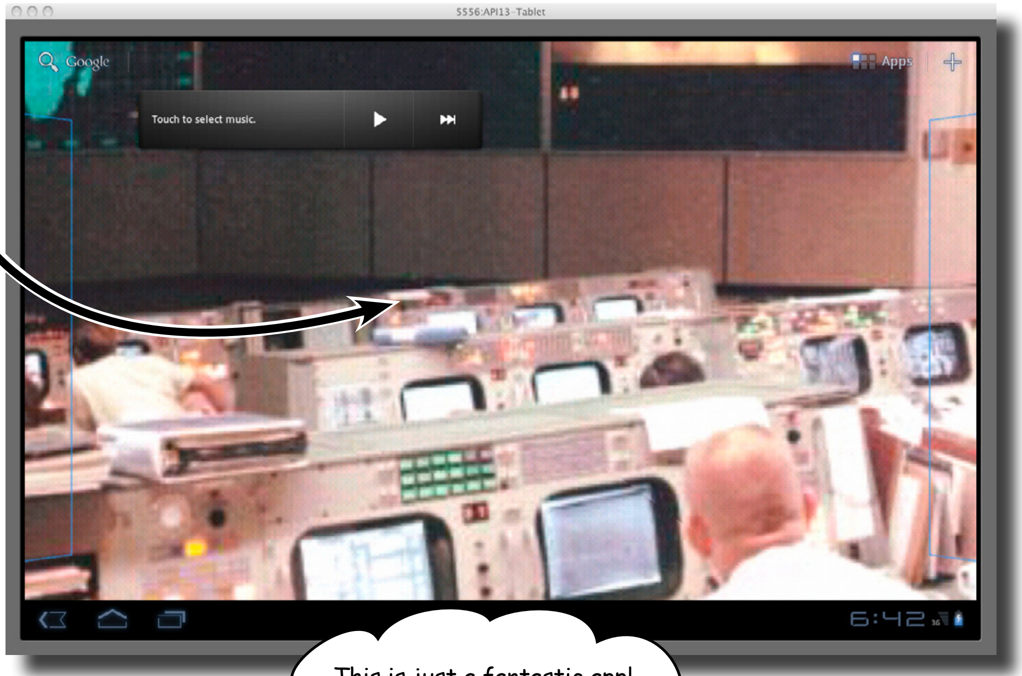
Test Drive

Now that you added the `onClick` methods to the `NasaApp Activity`, run the app again and see if the force close is resolved.



No errors! Now check the home screen to see if the wallpaper was set.

The wallpaper setting looks great!



This is just a fantastic app! Cool images, optimized for tablets in various sizes. I'm impressed!



Brilliant work

You've got the complete Nasa Image of the Day app is working great. The tablet app is running with Fragments so you can easily add and remove on screen content based on screen size. And this was on top of the already super customized layout work you did in Chapter 5. This is one tuned app!



Go Off Piste

That was some great work you did Fragments in this chapter, and all of the other work on the Nasa Image of the Day. Here are some ideas for additional exploration.

New Activities for Small Devices

In this chapter, you only showed the Image of the Day fragment for small devices. But the Education News can be interesting! After reading chapters 7–9 and learning more about creating additional Activites in your app, create a second Activity using the same two fragments. Then you should be able to show both fragments in one Activity or one Fragment each in two different Activities.

Explore Additional Overrides

In addition to screen size and pixel density, you can customize layouts based on device hardware form factors. Try building a custom layout for a specific form factor (like a device without touch screen support). Then create an AVD for that configuration. Test that your override works and doesn't effect other form factors.

Refresh both Fragments

The refresh button only worksfor the Image of the Day Fragment. After moving the buttons out of the Image fragment, make the refresh button refresh both Fragments.

Move the buttons

The refresh and set wallpaper buttons look a bit cluttered in the fragment on screen. Move them to a button bar, or as menu items (after you learn about them later in the book).



Your Android Toolbox

Now that you're getting a handle on optimizing for tablets, you can build all of your apps with tablet support!

Converting to Fragments

- Extend fragment instead of Activity
- Call getActivity() before any Activity method you called in the Activity you're converting
- Update onClicks and other mechanisms relying on direct access to an Activity
- Make any layout updates needed for the fragment to layout correctly inside another view since it will no longer be full screen

New Screen Configuration

- Put layouts in folders by minimum screen width in DPs. For example, layouts in /res/w720dp will load if the screen is at least 720 dp wide
- Also put layouts in folders by minimum screen height. Layouts in /res/h1024dp will load if the screen is at least 1024 dp high.
- You can also use smallest width which combines the two. So layouts in /res/sw600dp will only load if both the width AND the height are greater than 600dp.

BULLET POINTS

- Install new Android versions as needed using the **SDK and AVD Manager**
- Make new **AVDs** for new versions
- Set the Android version for your project in Project Preferences
- Combine multiple `Activities` on one screen using `Fragments`
- Convert existing `Activities` to `Fragments`, or write new `Fragments` from scratch
- Override `Fragment` lifecycle methods as needed, specifically `onCreate`, `onCreateView` and `onStart`
- Return the `Fragment` view in `onCreateView`, don't set the content view from a `Fragment`
- Inflate layouts with `LayoutInflater`
- Add `Fragment` in layouts (or in code). Then the layout is inflated, your fragment will be automatically created, started and connected to the launching `Activity`.
- Fragments are supported back to Android 1.6. View the Android Compatibility Package for more information: `http://developer.android.com/sdk/compatibility-library.html`.

7 lists and adapters

Building a list-based app



Where would we be without lists? They display read-only information, provide a way for users to select from large data sets, or even act as navigational device by building up an app with a list-based menu structure. In this chapter, you'll learn how to build an app centered around a list. You learn about Adapters where lists store their data, and how to customize the data rendered in your list.

Donna is training for a big race...

Donna jogs all the time, but she hasn't raced before. There is a big race coming up and she wants to be in super shape to get a great time.

Donna knows the only way to improve is to train consistently and track her progress over time, constantly improving any issues. She wants to track her progress on her Android phone since she always has it with her. But she doesn't like any of the apps she's found.

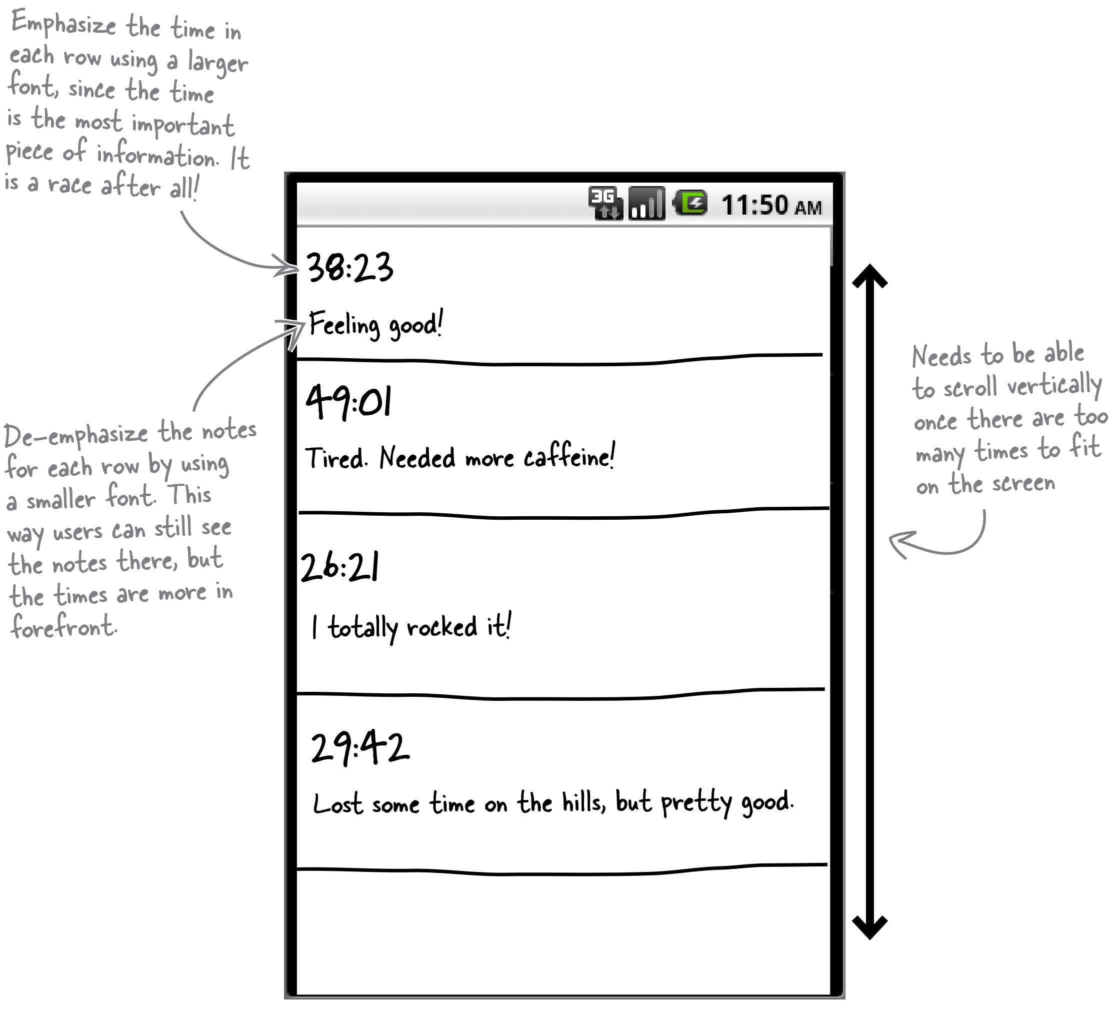


All of the apps she's found are too complicated.

She's found lots of tracking apps, but they all use GPS, have subscriptions, or just make things *too complicated*. Donna asked you to build the simple time tracker app for her, and as a good friend how could you say no!

Time List Screen Construction

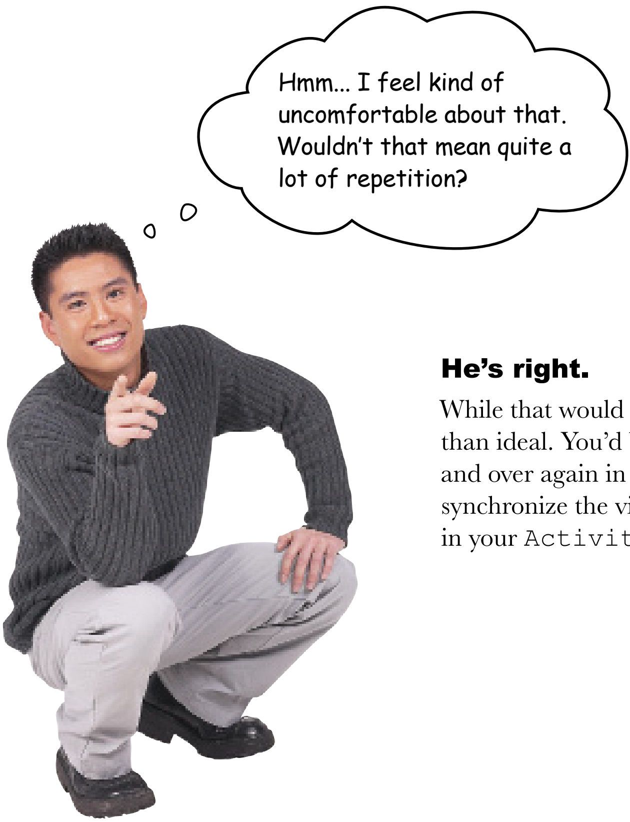
Donna gave you this sketch for the time tracker app's time list screen. It's a pretty simple screen with the list of times and notes, just like she said.



But which View should you use to implement this sketch?

Plan the implementation

You have a pretty clear sketch of the app to build, and now you need to decide how you're going to implement it. You *could* create a `LinearLayout` and add Views dynamically based on the items to be displayed and then put that `LinearLayout` in a `ScrollView`.



He's right.

While that would technically work, it seems a little less than ideal. You'd be repeating the same layout over and over again in the list and you'd have to somehow synchronize the views on screen with the data stored in your `Activity`. But what's the alternative?

Geek Bits

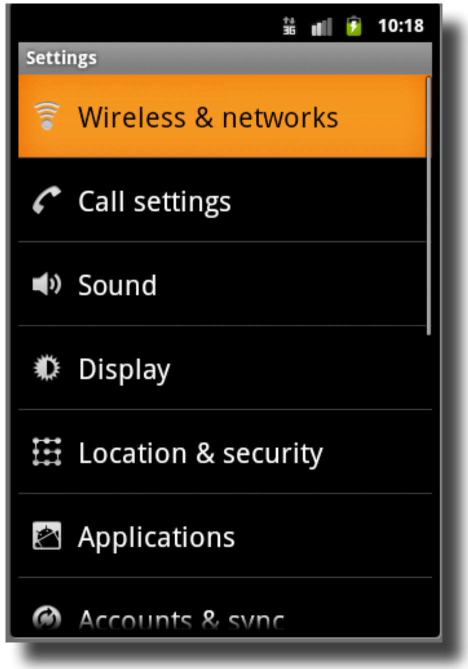
You can get a reference to a ViewGroup using findViewById. Once you have the ViewGroup reference in code, you can programatically add Views to that ViewGroup at runtime. This isn't done in any of the book examples, but it can be really useful way to declare most of your layout in XML and add a bit of dynamic behavior.


Wouldn't it be dreamy if there were a built in way to manage lists of information. But I know it's just a fantasy…

Use ListView

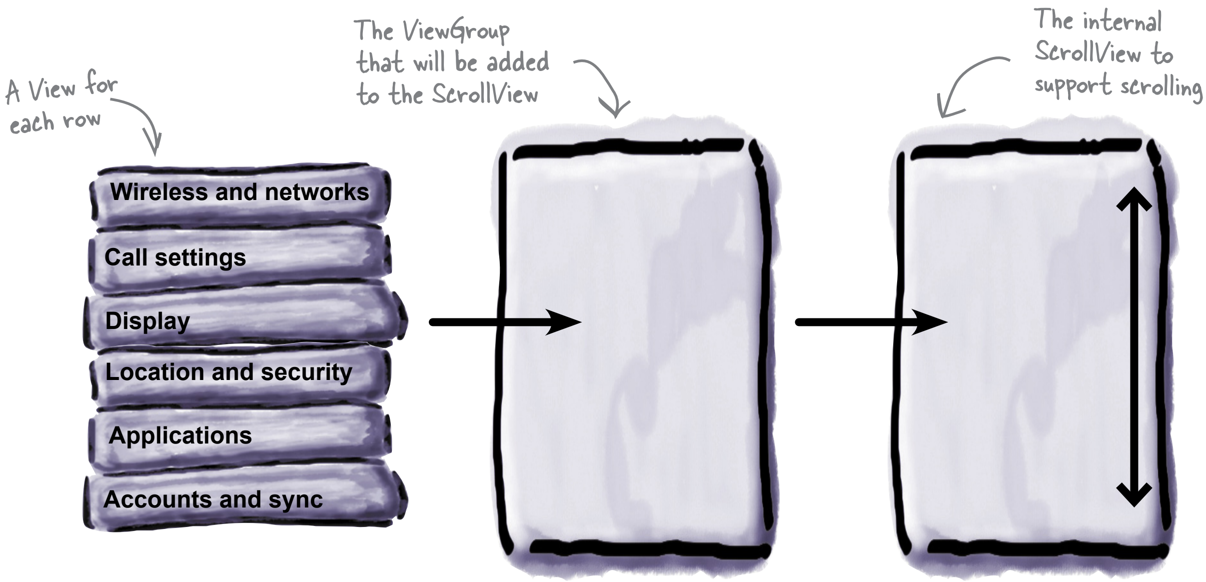
ListView is a built in Android View that displays items in a vertical list. It has built in functionality for most of what you'd want a list to do- like automatically scrolling when the screen is filled with data, as well as a clean way to separate your data displayed in the list from the ListView itself.

You'll find ListViews all over Android. Here's an example of a ListView used in the About Phone screen from the Android settings.



The many pieces of a ListView

ListView isn't just a View, it actually a complete ViewGroup on its own. A ListView contains Views for each of the rows, which are then added to a single ViewGroup and added to the ScrollView so the list can scroll. And this is all done internally inside the ListView. The end result is that a ListView is a *ViewGroup*, not just a View.



Add a ListView to your screen

Any `ViewGroup` (like a `LinearLayout` or the `ScrollView`, or even any `View`) can be added as the root element of the layout. And since you want to stretch the `ListView` fill the entire screen, the `ListView` is the one and only `View` you need in your layout. Add it to the layout in `main.xml` as the root `View` and adjust the width and height to fill the screen.

The entire layout for main.xml, which is the layout for TimeTrackerActivity

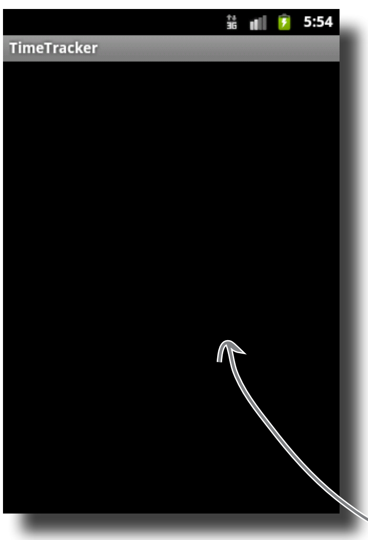
```
<ListView
      xmlns:android="http://schemas.android.com/apk/res/android"
      android:layout-width="fill_parent"
      android:layout-height="fill_parent"
      />
```

Don't forget to add the xmlns attribute since this is the root element of the layout

make the ListView stretch to the edges of the screen, both vertically and horizontally



Test Drive



The screen is empty, as nothing has been added yet.

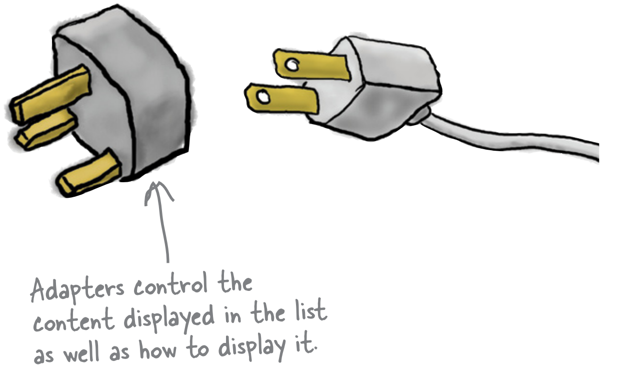
Run the app, and you'll see an empty screen. This isn't surprising since you added the `ListView`, but the `ListView` has no data to display yet.

Let's add some data to the list!

Lists are populated with data from adapters

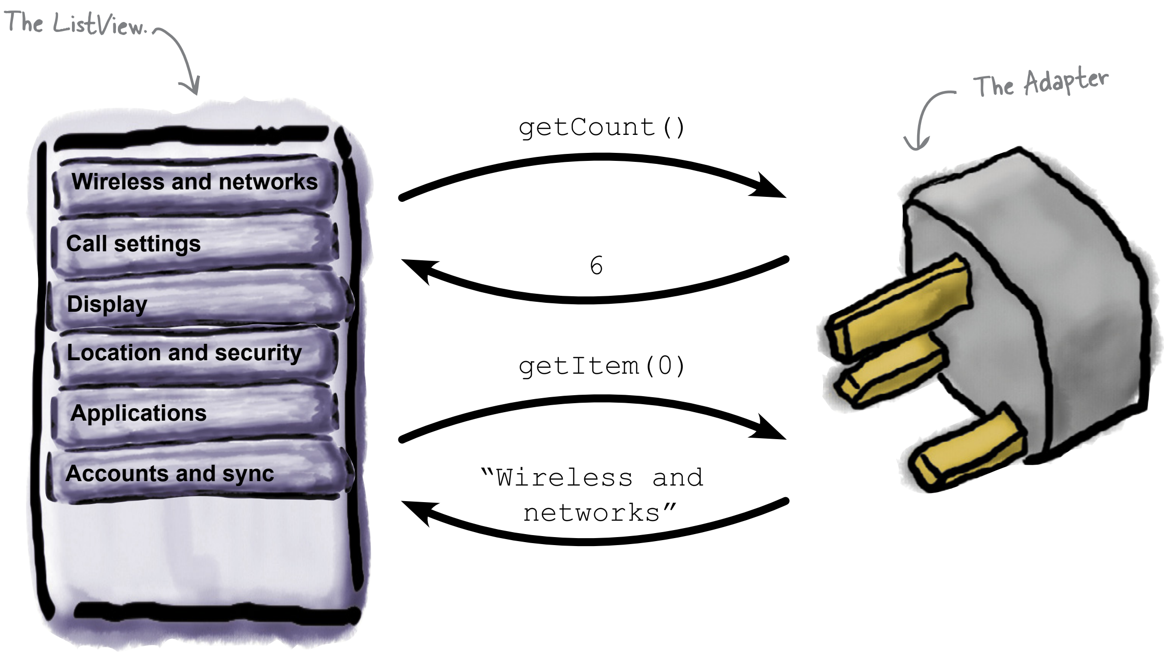
ListViews don't actually contain any data themselves. That's why you didn't see anything on the screen for the first **Test Drive**. The ListView was in fact on the screen, but it was empty so the screen appeared empty.

You can populate your ListView's with using an Android **Adapter**. Adapter is an interface whose implementations provide data and the display of that data used by the ListView. ListViews own Adapters that completely control the ListView's display.



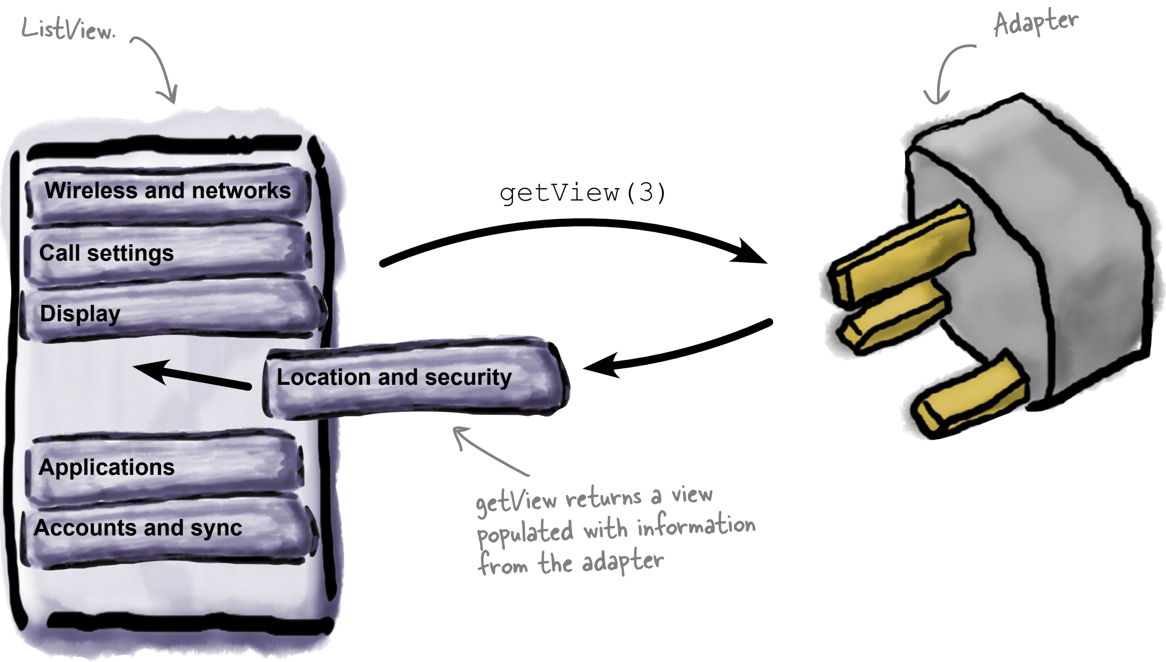
Communication methods

The Adapter interface includes a number of methods to communicate data to the ListView. This includes methods to determine how many elements need to be displayed, and to retrieve specific items.



Control methods

The Adapter interface also includes methods that control the display of that data like getView() that creates and populates a View that is displayed in the ListView.





Geek Bits

Android ListViews and Adapters are not clearly separated according to Model View Controller (MVC) lines. With MVC, you completely separate the data (the Model in MVC) from the display (the View in MVC) with communication facilitated by the Controller. However, Adapters perform Controller functions as well as some View and Model functions. This isn't a problem, and you can still properly organize and encapsulate your View and Model code. Just be aware that you won't *always* have the clear MVC separation you have in some other UI frameworks.

Build your own Adapter

You can populate your list with data by building your own `Adapter`. `Adapter` is an interface and you can implement your own from scratch.

Buy why build your own `Adapter` completely from scratch when there is a much easier way to go! Android provides an Abstract class called `BaseAdapter` that has most of the `Adapter` methods already implemented for you.

Start by creating a new class in your project called `TimeTrackerAdapter` and make it extend `BaseAdapter`.

TimeTrackerAdapter class before implementing the abstract BaseAdapter methods.

```
public class TimeTrackerAdapter extends BaseAdapter {
}
```



TimeTrackerAdapter.java

there are no Dumb Questions

Q: Do I have to use BaseAdapter?

A: No, you can write your own Adapter implementation from scratch if you choose.

Q: When would I want to do that?

A: There are a number of different reasons you may want to write your own. BaseAdapter handles a lot of the Adapter implementation for you, but if you want something custom or extremely optimized for your app, you may need to write your own.

Q: Is there any downside to writing my own Adapter?

A: Writing your own Adapter is completely fine. But it does take some work to rebuild what you get for free with BaseAdapter. Plus, if you use BaseAdapter, the BaseAdapter implementation could be improved pver time. And if it is improved, you'll get that benefit for free too.

Q: So is it a good idea to use BaseAdapter?

A: For the most part, use BaseAdapter unless you have a good reason **NOT** to.

Implement the abstract methods

Now implement the abstract `BaseAdapter` methods. The easiest way to do this in Eclipse is to go to the Eclipse menu and select Source → Override/Implement methods.

TimeTrackerAdapter class after implementing the abstract BaseAdapter methods.

```
public class TimeTrackerAdapter extends BaseAdapter {
       public TimeTrackerAdapter() { }

       public int getCount() {
              return -1;
       }

       public Object getItem(int index) {
              return null;
       }

       public long getItemId(int index) {
              return -1;
       }

       public View getView(int index, View view,
              ViewGroup parent) {
              return null;
       }
}
```

Eclipse will fill in auto-generated implementations like these

There are three data related methods you need to implement in BaseAdapter subclasses.

There is just one view mthod you have to implement in BaseAdapter subclasses... the method that returns the view used to display data in the ListView.



TimeTrackerAdapter.java

Building out the adapter

Now you have a `BaseAdapter` implementation, but it still doesn't store any data for your list. It's just filled with autogenerated methods that will compiles, but don't do anything useful yet.

Here's what you;re going to do to make this adapter work for you!

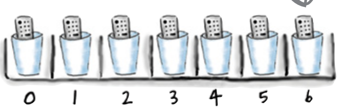
1. Create a data object

Based on the app design, you'll need to store a time and note for each time entered. Rather than separately storing that information, create a data object to store both fields in a single object.



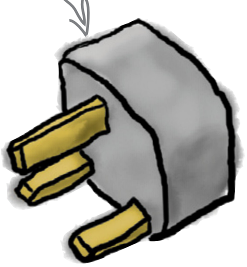
2. Add an ArrayList of data objects

Now that you have the data object for a single time record, add an `ArrayList` to store these data objects in your `Adapter`.



3. Complete the adapter methods

Now that you have the `ArrayList` of your data objects, you can finish implementing the `Adapter` based on the data stored in the `ArrayList`.



Create a data object

Start by creating the data. Since it's an object storing all of the information for a specific time, call it `TimeRecord`. It should have two variables, one for the time and one for the notes. Add a constructor, getters, and setters for both variables.



Ready Bake Code

```
public class TimeRecord {
      private String time;
      private String notes;

      public TimeRecord(String time, String notes) {
            this.time = time;
            this.notes = notes;
      }

      public String getTime() { return time; }
      public void setTime(String time) { this.time = time; }
      public String getNotes() { return notes; }
      public void setNotes(String notes) { this.notes = notes; }    }
```



TimeRecord.java

Do this!

> Create the TimeRecord class in your project. Add the the code above to the new class.

Sharpen your pencil

Below is the `TimeTrackerAdapter` code with the autogenerated methods Eclipse created when you implemented the `BaseAdapter` methods. Using the `TimeRecord` data object, complete the data methods `getCount`,and `getItem` (`getItemId` is done for you). You'll also need to create a collection to store these objects.

```
public class TimeTrackerAdapter extends BaseAdapter {

    public TimeTrackerAdapter() {
    }

    public int getCount() {
        return -1;
    }

    public Object getItem(int index) {
        return null;
    }

    public long getItemId(int index) {
        return -1;   return index;
    }

    public View getView(int index, View view, ViewGroup parent) {
        return null;
    }
}
```

Add a collection to store TimeRecords as a member variable.

This just needs to return a unique ID for the data. And since the index ID is unique for a row, standard practice is just to return the index. (The line `return -1;` is struck through and replaced with `return index;`.)

Ignore getView for now. You'll implement that method after finishing the data methods.



TimeTrackerAdapter.java



Adapters Exposed

This week's interview:
Combining your Data and Display: Good or Bad?

Head First: Hi Adapter, thanks for joining us!

Adapter: Always a pleasure.

Interviewer: Let me get right down to business. Most user interface frameworks are pretty serious about very clear Model View Controller (MVC) separation, but not you.

Adapter: What can I say? I'm a renegade.

Head First: Aren't you afraid the **Design Pattern Police** are going to come after you?

Adapter: One step ahead of you! I was worried people would start clamoring about how I'm not pure MVC and all that, so I changed my name. I'm not a Model, Controller, View, or any combination of them. I'm my own Object. That's why I'm called ***Adapter***.

Head First: Fair enough. Do you find it confusing to have all of that logic for data and views in your implementations?

Adapter: Not really. Most of the time, the data I'm storing is directly related to me and why I'm on a screen in the first place. Maybe I'm displaying a list of States for selection in an Address entry process or maybe I'm displaying read only data like the times in the TimeTracker app. I can also be used as a navigation device with nested menus. Most of the time my data storage is pretty minimal and directly related to displaying it. Really, it just makes sense to keep it together.

Head First: Sometimes it must get confusing though, right?

Adapter: Absolutely! If I'm displaying a huge list of information that's stored elsewhere (say in a database on the phone) I don't want to bloat myself by storing that data inside me AND in the database. That would be wasteful.

Head First: And what do you do then?

Adapter: Well, there is nothing saying I have to store the data in me! I just have to facilitate providing that data to the ListView. I could easily lop off a piece of myself and turn that into a pure data source. As long as I have a reference to that new data source, I can ask it anything that the ListView asks me- how many rows, the data for a row and so on.

Head First: I like the fact that you can still separate out your data and provide it to the ListView. That lopping off bit sounds painful though!

Adapter: Oh, it's not so bad.

Head First: And there you have it. Adapter, the renegade MVC recluse, with the ability to control it's own data and display. Thanks for joining us!

Adapter: My pleasure! Thanks for having me.

Sharpen your pencil Solution

Below is the `TimeTrackerAdapter` code with the autogenerated methods Eclipse created when you implemented the `BaseAdapter` methods. Using the `TimeRecord` data object, you should have completed the data methods `getCount`,and `getItem` (`getItemId` is done for you). You also should have created a collection to store these objects.

```
public class TimeTrackerAdapter extends BaseAdapter {
        private ArrayList<TimeRecord> times = new ArrayList<TimeRecord>();
        public TimeTrackerAdapter() {
        }

        public int getCount() {
                ~~return -1;~~   return times.size();
        }

        public Object getItem(int index) {
                ~~return null;~~   return getItem(index);
        }

        public long getItemId(int index) {
                ~~return -1;~~   return index;
        }

        public View getView(int index, View view, ViewGroup parent) {

                return null;
        }
}
```

A private ArrayList containing one TimeRecord for each row in the ListView.

Since there is one TimeRecord for each row, the size of the ListView is just the number of TimeRecords in the ArrayList.

Again, the one-to-one mapping keeps everything easy! The data for a row at the index is the TimeRecord in the ArrayList at that same index.



TimeTrackerAdapter.java

What about getView?

The data methods are complete now, but what about `getView`? The `getView` is the link between the data stored in the `Adapter` and how it's displayed in the `ListView`. In the `getView` implementation, you'll retrieve the data for the row from the ArrayList, populate a view with that data and return the populated view.

```
public View getView(
        int index,
        View view,
        ViewGroup parent)
```

The index of the data to display. This corresponds to the indeces in the array list of TimeRecords.

The view to populate the data in.

Hold on a second. What view is going to be used here? Don't you have to customize one for time tracker data?

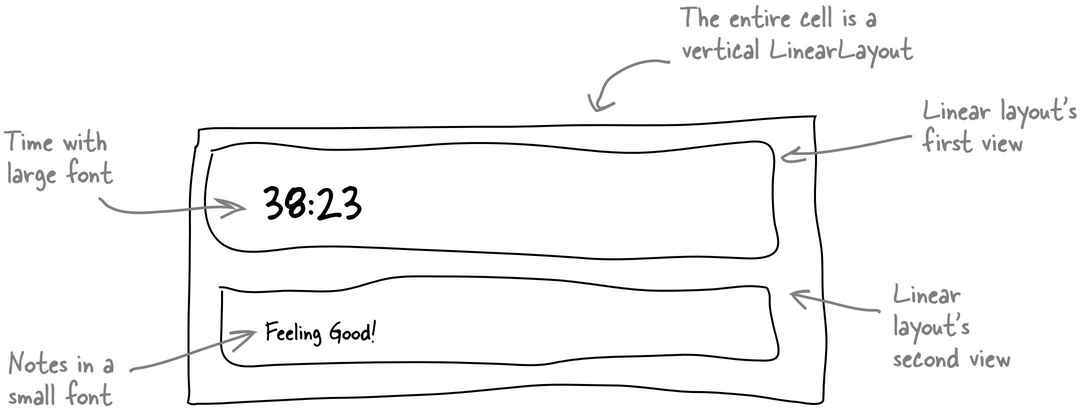


You'll need to create a custom view

You're storing custom data for you app in the Adapter. That's why you had to subclass BaseAdapter and create your own implementation. Just like storing your custom data, you also need to create your own custom views to display your data.

Adapter View Construction

Before you can wire up the `View` to the `TimeRecord` in `getView`, you need to design it! Here is a sketch of the layout for one row in the `ListView`.



* The widths are all set to FILL_PARENT so they are as wide as possible.
The heights are set to WRAP_CONTENT so they can resize based on contents.

there are no Dumb Questions

Q: How come all of the View height are set to wrap_content?

A: First of all, if you set the height to fill_parent, it will fill the whole list! That's very bad! Likewise, setting the height to a fixed size would be a poor choice. The time is fixed in length to one line, but the notes could be several lines. but if you set the height of each View and the layout to wrap_content, the cell will grow to fit the content and the row data will display correctly (and completely!).

Q: We're making a new layout here, can you have more than one layout per screen?

A: Definitely. Up to now, you've had exactly one layout for each Activity (which maps to a screen). It doesn't have to be that way! You can have as many (or as few) layouts per screen as you like.

Q: That sounds kind of cool, when would I want to use a lot of layouts?

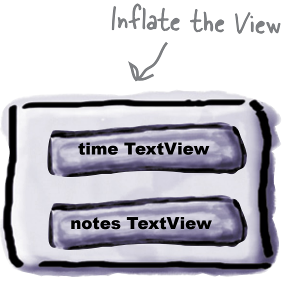
A: Well, custom ListView rows are obviously one example but there are more. There is a really cool technique where you can use the <include> directive in one of your layouts. That takes another layout that you're including and adds it inline making a big combined layout. It's a really useful way to organize and encapsulate complicated layouts. We don't have time to go over it in this book, but it's definitely something worth checking out in the online Android documentation.

The steps to complete getView

The `getView` method does some serious heavy lifting for the Adapter. It's really the method that bridges the gap between the data and the display. Not surprisingly, there are a few basic tasks you'll need to accomplish inside every `getView` implementation.

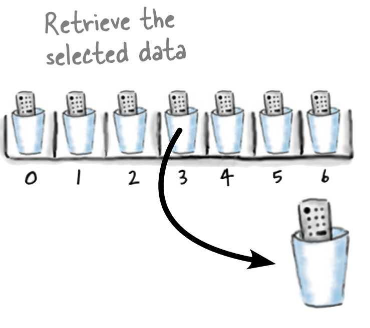
Instantiate the View

The first time `getView` is called on your Adapter, the View passed in is null. Since the Adapter knows how the data should be displayed, it's up to the Adapter to instantiate the View the first time. Successive calls to `getView` return the same View back to be repopulated with new data. Repopulating the same View instead of creating new Views for every cell is a performance optimization often used in user interface frameworks.



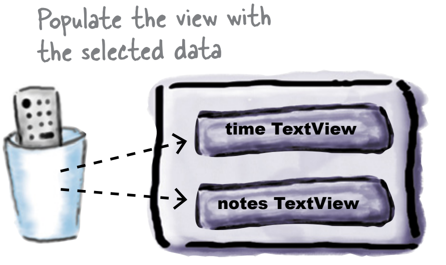
Retrieve the data

The Adapter also contains the data. And the list index is passed into `getView`. You'll need to correlate the index passed in to the ArrayList of `TimeRecords`. For this adapter, you have an correlated indices between the ListView and the `TimeRecords` in the `ArrayList`.



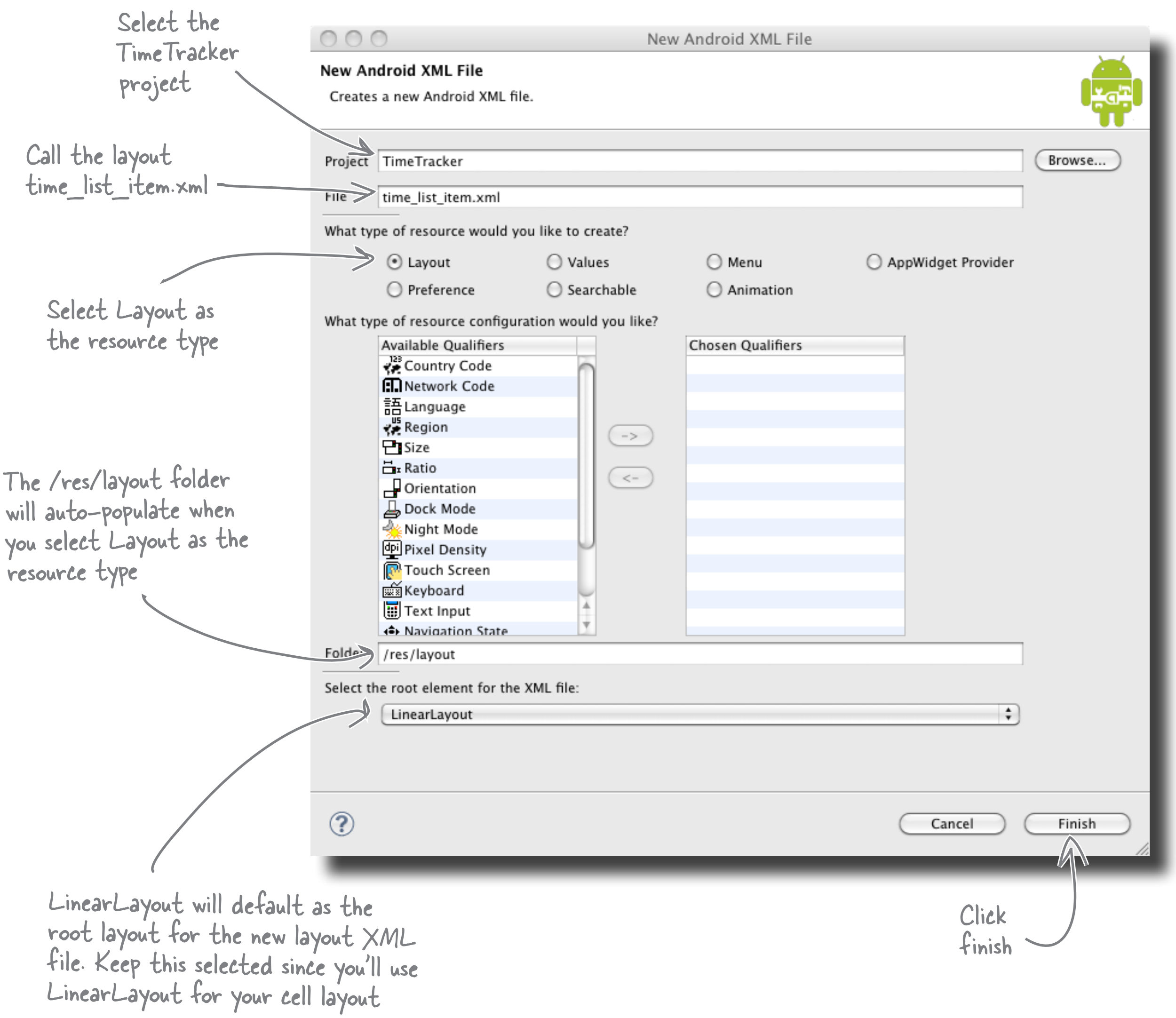
Set values on the view

Using the selected `TimeRecord`, and the View, set properties on the view to reflect the data. In this case, you'll be setting text in the view to display the time and the notes from a `TimeRecord`.



Create the new layout

Now that you have a design for your view, it's time to build it! Go to File → New → Android XML File to launch the new Android XML file wizard.



Sharpen your pencil

Below is the time_list_item layout code generated by the New Android XML File Wizard. Modify the layout to match the design you created for the list cell.

```
<LinearLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" >
</LinearLayout>
```



time_list_item.xml

Sharpen your pencil Solution

Below is the time_list_item layout code generated by the New Android XML File Wizard. You should have modified the layout to match the design you created for the list cell.

```
<LinearLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" → "wrap_content"
</LinearLayout> android:orientation="vertical" >

    <TextView android:id="@+id/time_view"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:textSize="18dp"
            android:paddingBottom="5dp" />

    <TextView android:id="@+id/notes_view"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:textSize="12dp" />

</LinearLayout>
```

Set the height to wrap_content so it won't fill up the whole list.

Make sure the layout is Vertical.

The first TextView if for the time. It has an ID for access later from findViewById.

Make the width as wide as possible but size the height to the content

Make the text BIG!

Add some padding on the bottom so there is some space between the time and the notes.

The second TextView if for the notes. It also has an ID for access later from findViewById.

Make the width as wide as possible but size the height to the content.

Make the text small.

End the layout.

time_list_item.xml



Topic Title Magnets

Now that you've completed the View, you have everything you need to write the `getView` method. First you'll need to check and make sure the View is not null, and if it is null, you'll need to inflate it. Then you'll retrieve the selected `TimeRecord`. Once you've retrieved it, you need to populate the view with the information from that `TimeRecord`. Complete the `getView` method using the magnets below.

```
public View getView(int index, View view, ViewGroup parent) {









        return view;
}
```

Your magnets.

```
if (view == null) {
```

```
}
```

```
LayoutInflater inflater =
        LayoutInflater.from(parent.getContext());
```

```
notesTextView.setText(time.getNotes());
```

```
view = inflater.inflate(
        R.layout.time_list_item, parent, false);
```

```
TimeRecord time = times.get(index);
```

```
timeTextView.setText(time.getTime());
```

```
TextView notesTextView = (TextView)
        view.findViewById(R.id.notes_view);
```

```
TextView timeTextView = (TextView)
        view.findViewById(R.id.time_view);
```



Topic Title Magnets Solution

With the View completed, you had everything you needed to write the `getView` method. First you should have checked that the View is not null, and if it is null, you should have inflated it. Then you should have retrieved the selected `TimeRecord`. Once retrieved, you should have populated the view with the information from that `TimeRecord`, completing the code with the magnets.

```
public View getView(int index, View view, ViewGroup parent) {

    if (view == null) {
        LayoutInflater inflater =
                LayoutInflater.from(parent.getContext());
        view = inflater.inflate(
                R.layout.time_list_item, parent, false);
    }
```

Check if the View is null. If it is, retrieve the layout inflater and inflate the view.

```
    TimeRecord time = times.get(index);
```

The TimeRecord in the ArrayList at the index has everything you need to populate the view.

```
    TextView timeTextView = (TextView)
            view.findViewById(R.id.time_view);
    timeTextView.setText(time.getTime());
```

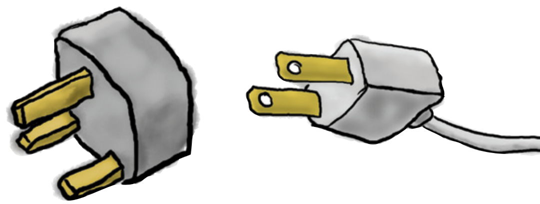
For the time, get a reference to the time TextView and set the text to the time String from the TimeRecord.

```
    TextView notesTextView = (TextView)
            view.findViewById(R.id.notes_view);
    notesTextView.setText(time.getNotes());

    return view;
}
```

Do the same process for the notes. Get a reference to the notes TextView and set the text to the notes String in the TimeRecord.

Connect the adapter to the ListView



The `Adapter` is finished now, and the next step is to use the `Adapter` in the `ListView`. To set the Adapter on the `ListView`, you'll get a reference to the `ListView` using findViewById and call the `setAdapter` method passing in an instantiated `TimeTrackerAdapter`.

Start by adding an android:id to the ListView in the layout.

```
<ListView
      xmlns:android="http://schemas.android.com/apk/res/android"
      android:layout-width="fill_parent"
      android:layout-height="fill_parent"
      android:id="@+id/times_list"
      />
```

Give the list an id.



main.xml

Now get a reference to the `ListView` in `onCreate`, instantiate the `TimeTrackerAdapter` and configure the `ListView` to use it.

```
public class TimeTracker extends Activity {
      TimeTrackerAdapter timeTrackerAdapter;

      public void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.main);

            ListView listView = (ListView)
                  findViewById(R.id.times_list);
            timeTrackerAdapter = new TimeTrackerAdapter();
            listView.setAdapter(timeTrackerAdapter);
      }
}
```

Get a reference to the ListView.

Instantiate the adapter.

Configure the ListView to use the adaapter.



TimeTracker.java

Add test code to the adapter

You custom `Adapter` implementation is now complete and being used in the `ListView`. There's just one problem. the Adapter still doesn't have any data in it.

You've built the `TimeRecords` data object to hold times entered, and built the `Adapter` around an `ArrayList` of `TimeRecords`. So even if you

```
public TimeTrackerAdapter() {
      times.add(new TimeRecord(
            "38:23", "Feeling good!"));
      times.add(new TimeRecord(
            "49:01", "Tired. Needed more caffeine"));
      times.add(new TimeRecord(
            "26:21", "I'm rocking it!"));
      times.add(new TimeRecord(
            "29:42", "Lost some time on the hills, but pretty good."));
}
```

Create a few prepopulated TimeRecord objects to see in the ListView.



TimeTrackerAdapter.java

Do this!

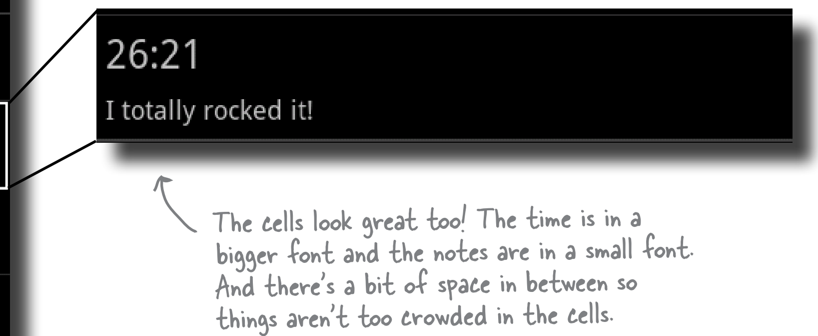
Add this test code to the constuctor of `TimeTrackerAdapter`.



Test Drive

Now that the `TimeTrackerAdapter` is complete, connected to `ListView` and populated with test data, run the app again and make sure it all worked!





The ListView has data now. Great work!

Donna's checking in...

Donna's really looking forward to using the app. So she stopped by to see how you're doing.



Next up, user entered times

In this chapter, you created the new project, added a list,build your own adapter, custom views, and connected it all together. And great work!

In the next chapter, you'll be adding a second screen to this app, so your users can enter their own times.

See you back shortly to add user entered times.



Go Off Piste

With all of this work wrapped on Adapter, you're ready to move on with this app. If you're still wanting to learn more about Adapter and their Views, here are a couple of places to look.

Prebuilt List Views

Although you built this list item view from scratch, sometimes you can use prebuilt views. Take a look at the constants in android.R.layout for more information: http://developer.android.com/reference/android/R.layout.html.

Built in Adapters

Take a look at these built in Adapters for your apps.

- ArrayAdapter: Adapter with everything implemented for you, just pass in an array!
- SimpleAdapter: Adapter that uses data stored in XML resources to build the list
- CursorAdapter: An adapter that uses information stored in a SQLite database (you'll learn more about these in a few chapters)



Your Android Toolbox

Now that you created an Adapter and list item View from scratch, you'll be able to add lists to all your apps.

Using ListViews

- Implement Adapter by subclassing BaseAdapter, writing your own, or using a prebuilt Adapter.
- Create an list item View or use a built in View.
- Populate the adapter with data.
- Configure the list to use your adapter.

BULLET POINTS

- When working on a multi-screened app, always start with your post important use case. (Talk to your users to find out what they are!)
- Use `ListView` to display information in a vertically oriented list (with built in scrolling!).
- Fill your lists with data using `Adapters`.
- Start your custom `Adapters` implementations using `BaseAdapters`.
- Use Eclipse's built in "Override/Implement Methods" option to add method stubs to your class for any interface your implementing (or abstract class you're extending).
- If you build an `Adapters` that stores data, build your own data object to keep your data organized
- Add new layouts to your apps using the Android New XML File Wizard
- Inflate layout XML descriptions into instantiated views using `LayoutInflater`.

8 multi-screen apps

Navigation in Android



Eventually, you'll need to build apps with more than one screen. So far, all of the apps you've built have only one single screen. But the great apps you're going to build may need more than that. In this chapter, you'll learn how to build an app with a couple screens, and you'll create a new Activity and layout, which the Wizard previously did for you. You'll learn how to navigate between screens and even pass data between them. You'll also learn how to make your own Android context men- the menu that pops up when press the Menu button!

Donna wants to enter her times

Donna thinks the app is looking great, and she's really looking forward to using it. But right now she can't enter her own times.



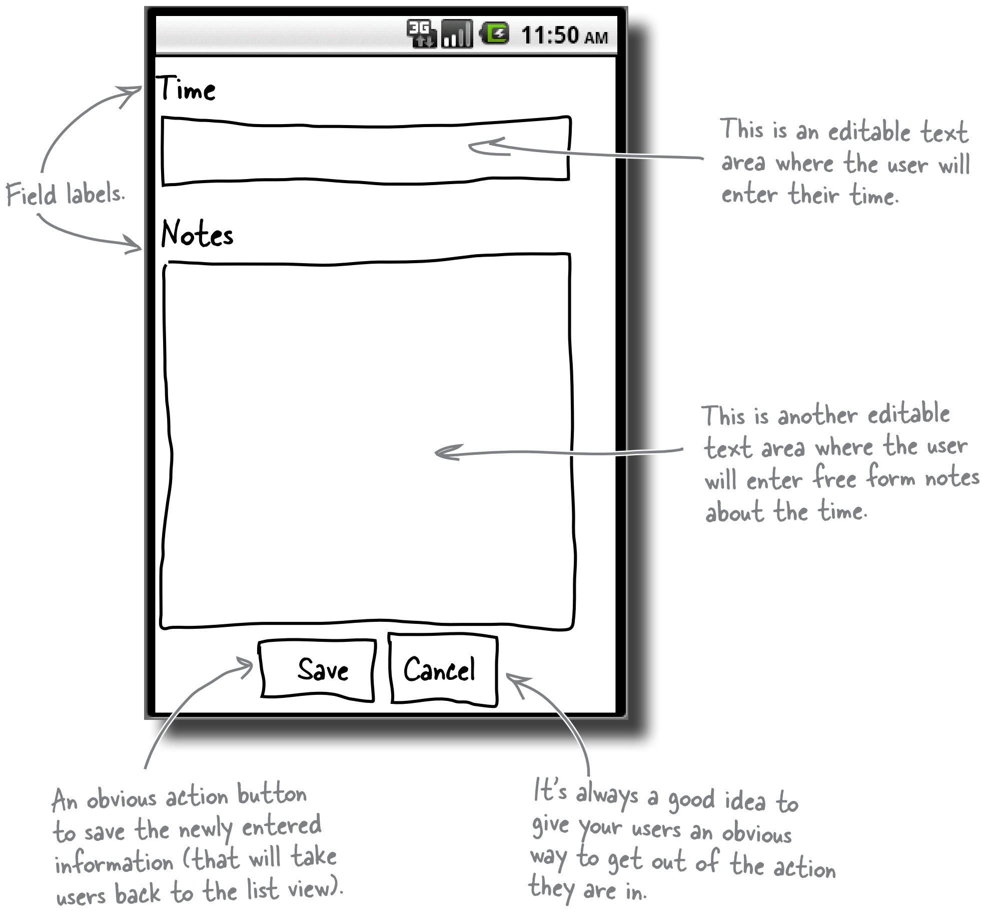
Let's get right on it!

The only thing stopping Donna from using her perfect new time tracking app is that she can't enter her own times yet. Let's build that now so she can get started tracking her times for her big race!

How is she going to add her own times?

The list is displaying times, and you need to make a way to add times with notes inside the app. You could combine it all into one screen and have an entry section at the bottom, but that would get cluttered very quickly.

The best way to do this is to add another screen specially designed for entering data. Here's a quick sketch of what the new screen will look like.

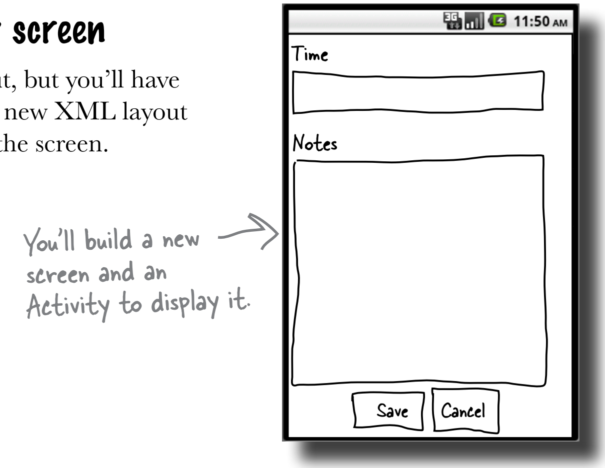


Adding the entry screen

There are a few steps you'll need to take to make the new entry screen and connect it to the list screen. Here is what you'll be doing in this chapter.

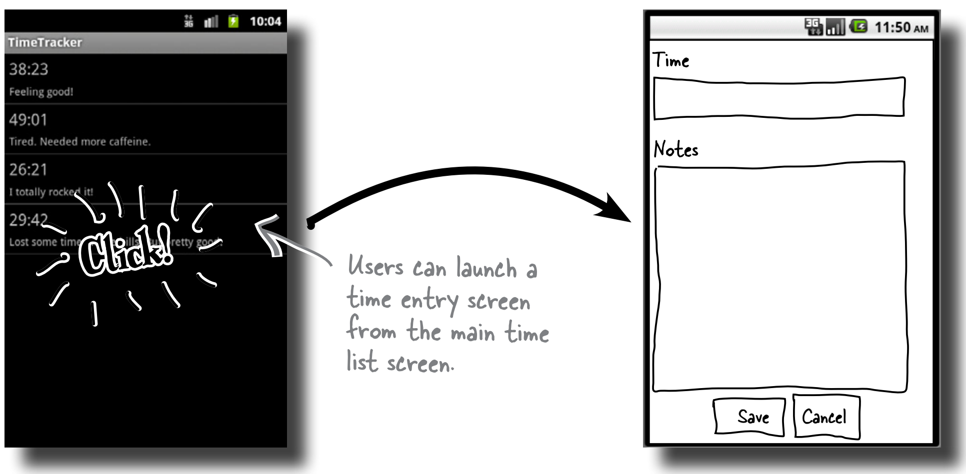
1. Build the new entry screen

The new screen is sketched out, but you'll have to build it. You'll be making a new XML layout and a brand new Activity for the screen.



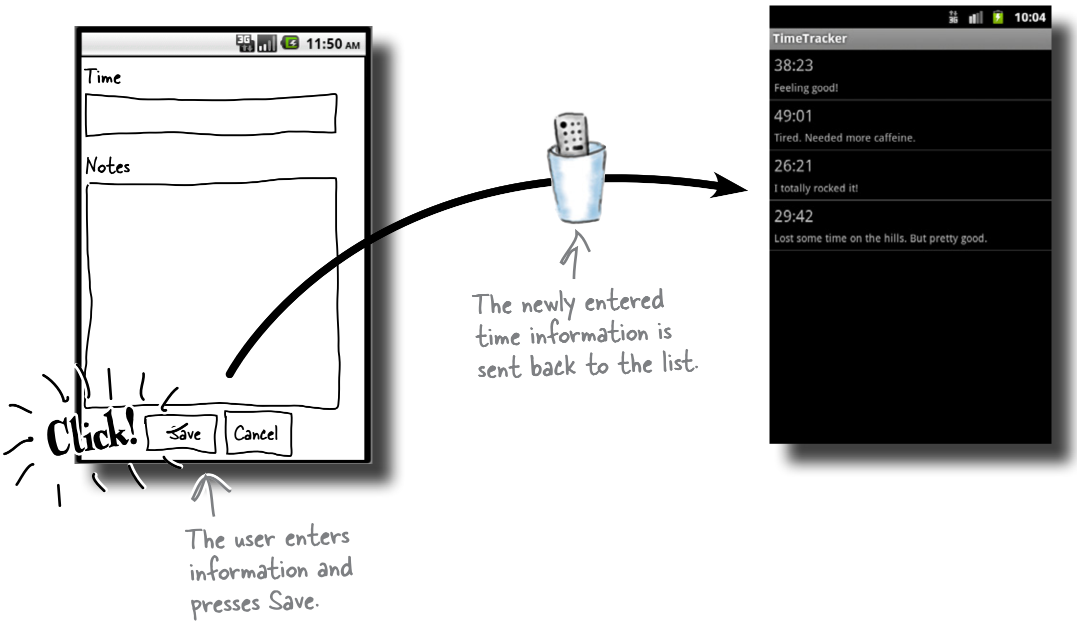
2. Launch the entry screen from the list

The list screen is the main screen for this app and this is the screen that displays when you launch the app. You'll add an menu with an 'Add' menu item to this screen that will launch the entry screen.



3. Return to the list screen from the entry screen

Whether the user enters a new time or cancels out of the entry screen, they need to return the list screen when they are done. After writing the code to navigate to the entry screen, you'll write the code to return back to the list screen with the user entered data.



4. Display the new time in the list

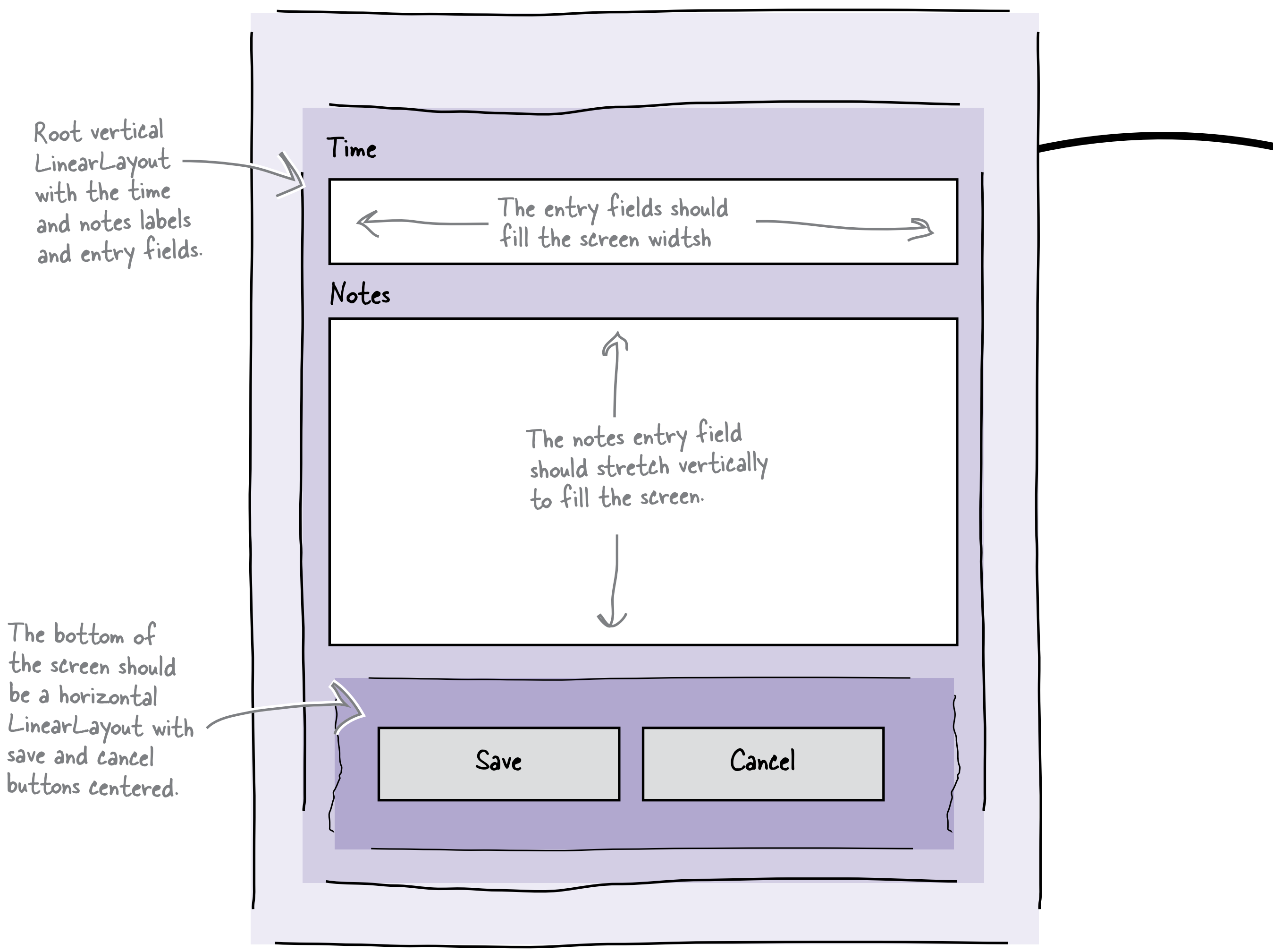
This is where it all comes together! After building the navigation back and forth from the entry screen, you'll implement logic to store the newly entered time and display it in the list.



Create the new layout xml file

Launch the **New Android XML File** wizard and create a new layout. Call the new layout `add_time.xml`.

Here is the plan for the layout. You'll create one *vertical* `LinearLayout` for the screen. This will have "Time" label, the text entry field to enter the time, followed by the "Notes" label and the notes entry field. At the bottom of the screen, you'll have a *horizontal* `LineatLayout` with the save and cancel buttons centered.



Use EditText for text entry

This is the first time you're adding a text entry component to one of your screens. All of the other Views you've added to your screens have been read only. But now you're having users enter information, so they need an entry View.

There is a special text entry View called EditText that you can use. It works just like a `TextView`, only it's editable. From a layout perspective, just remember to give the EditText an ID so you can retrieve the View and it's contents later on.

Root vertical LinearLayout with the time and notes labels and entry fields.

```
<EditText android:id="@+id/your_id"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      />
```

You can apply View layout attributes to an EditText just like other Views.

there are no Dumb Questions

Q: The New Android XML File wizard is pretty cumbersome. Do I have to use it to make new layout XML files?

A: No. The wizard is just creating the XML file and adding it to correct directory based on the XML type. It also tries to add a little structure based on your XML file type like adding the root element of a LinearLayout if your making a layout file that you've declared in the wizard to be a LinearLayout.

Q: After all that time customizing layouts for different screens in the NASA app, how come we're only adding one layout for this screen?

A: Just like the NASA app, you would want to test this app on multiple devices of various screen sizes and customize the layouts as necessary for your supported device.



Long Exercise

Below are magnets with the XML layout declarations for the `Views` in your layout. Arrange the magnets to complete the layout XML. There is one main layout and one sublayout for the button bar similar to the one you made for the NASA Daily Image app.

```
<EditText android:id="@+id/notes_view"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:gravity="top"
        android:layout_weight="1"
        android:layout_marginBottom="10dp" />
```

```
<LinearLayout
        android:orientation="horizontal"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_weight="0"
        android:background="#FF8D8D8D"
        android:gravity="center_horizontal" >
```

```
<TextView android:text="Notes"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
```

```
<TextView android:text="Time"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="10dp" />
```

```
<EditText android:id="@+id/time_view"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_marginBottom="10dp" />
```

```
<Button android:text="Save"
        android:onClick="onSave"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
```

```
<Button android:text="Cancel"
        android:onClick="onCancel"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
```

```
</LinearLayout>
```

```
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:orientation="vertical">
```

```
</LinearLayout>
```

Long Exercise Solution

Below are magnets with the XML layout declarations for the `Views` in your layout. You should have arrange the magnets to complete the layout XML. There is one main layout and one sublayout for the button bar similar to the one you made for the NASA Daily Image app.

This is the layout root, a vertically oriented LinearLayout for the screen.

```
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:orientation="vertical">
```

The time label.

```
<TextView android:text="Time"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:layout_marginTop="10dp" />
```

The time EditText. Notice it has an ID for later retrieval.

```
<EditText android:id="@+id/time_view"
       android:layout_width="fill_parent"
       android:layout_height="wrap_content"
       android:layout_marginBottom="10dp" />
```

The notes label.

```
<TextView android:text="Notes"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:layout_marginLeft="10dp" />
```

The notes EditText. Notice it also has an ID.

```
<EditText android:id="@+id/notes_view"
       android:layout_width="fill_parent"
       android:layout_height="wrap_content"
       android:gravity="top"
       android:layout_weight="1"
       android:layout_marginBottom="10dp" />
```

The inner linear layout for the button bar. It has a gray background and the gravity is set to center_horizontal so the buttons will be centered.

```
<LinearLayout
        android:orientation="horizontal"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_weight="0"
        android:background="#FF8D8D8D"
        android:gravity="center_horizontal" >
```

The save and cancel buttons which both have onClick properties defined. The methods will be implemented later.

```
    <Button android:text="Save"
            android:onClick="onSave"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" />

    <Button android:text="Cancel"
            android:onClick="onCancel"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" />
```

```
</LinearLayout>
```

End of the button bar layout.

```
</LinearLayout>
```

End of the screen.

Create a second Activity

Now that you have the layout built for the entry screen, you need to display it in the app. So far, you've displayed a layout when an Activity is created, you've created optimized layouts that dynamically display for different screen sizes, and displayed layouts as part of a fragment.

But now you're making an entirely new screen with new behavior. What you need now is another Activity. Start creating a new Activity by adding a Java class called `AddTimeActivity` to your project that extends `Activity`.

Make sure to extend Activity.

```
public class AddTimeActivity extends Activity {

}
```



AddTimeActivity.java

there are no Dumb Questions

Q: I already have an Activity. Do I really need another one?

A: In this case, yes. You could have displayed the new layout in the TimeTracker Activity, but that Activity has functionality specific to the list screen, like finding the list view in the layout and setting the adapter. If you just tried to display the entry layout in the TimeTracker Activity, the Activity would break when trying to find the list.

Q: When would be a good example of when I would have multiple layouts in one Activity?

A: The layout optimizations you did in chapters 5 and 6 for different devices consisted of creating multiple layouts for one Activity. The key is that the functionality and behavior were the same. In the NASA app, once you had different behavior for the NasaEdNews, you had a second Activity. Just remember, same behavior, same Activity. Different behavior, different Activity.

Do this!

Create a new class called `AddTaskActivity` in your project. Make sure it extends `Activity`.



Below is the code for the `AddTimeActivity` class you just created. Complete the code below to display the screen. You'll need to override `onCreate` and set the content view to your new layout.

```
public class AddTimeActivity extends Activity {






}
```

Override onCreate here. In that method, write the code to display the layout for the add task screen.



AddTimeActivity.java

Exercise Solution

Below is the code for the `AddTimeActivity` class you just created. You should have complete the code below to display the screen. You should have overrided `onCreate` and set the content view to your new layout.

```
public class AddTimeActivity extends Activity {

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.add_time);
    }

}
```

Dont forget to call super. (points to `super.onCreate(savedInstanceState);`)

Call setContentView with the R constant for the layout you just wrote to set the screen. (points to `setContentView(R.layout.add_time);`)



AddTimeActivity.java



Don't forget to call super.onCreate()

The Activity base class has logic needed to properly instantiate and configure an Activity for use by the Operating System. If you override one of the lifecycle methods, be sure to call super. If you don't you'll get a nasty runtime exception and your activity won't run!



There's work left to do, but you're getting there!

So far, you've built the layout for the new ttime entry screen and the Activity to control the screen's behavior.

Now it's time to navigate to the new entry screen from the list.

BRAIN POWER

Think about different Android apps you've used and how you navigate around them. How would you build the navigation to the Add Time screen in this app? Write your answer below.

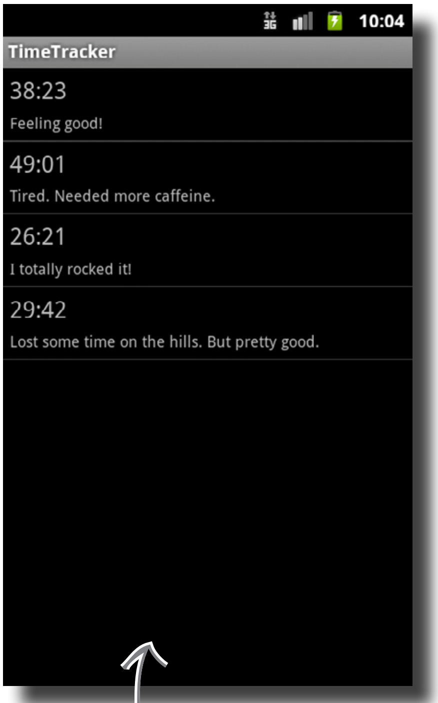
Use an Option Menu

With the layout built and a new Activity created for the Time Entry screen, it's time to navigate to it. There are a few different ways you could implement the navigation including putting a button on the screen or using an options menu.

The options menu is the popup that displays when you press the Menu button on an Android device (or the on screen menu button on a tablet). The options in the menu are controlled by the Activity in focus when the menu button is pressed.

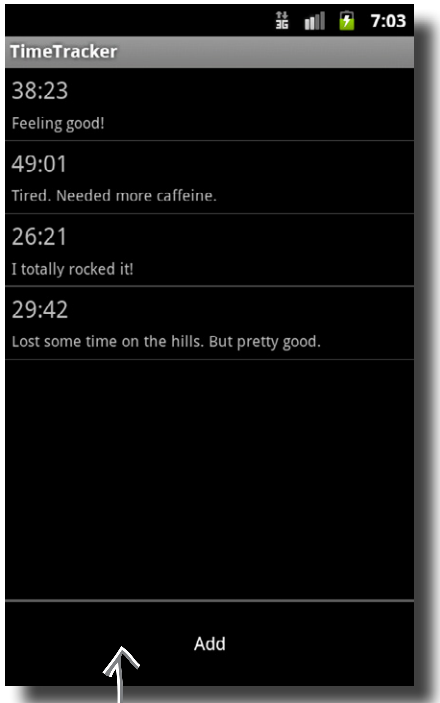
Let's add an options menu item to launch the time entry screen.

Options menu hidden



The list screen remains unchanged when the menu is not open...

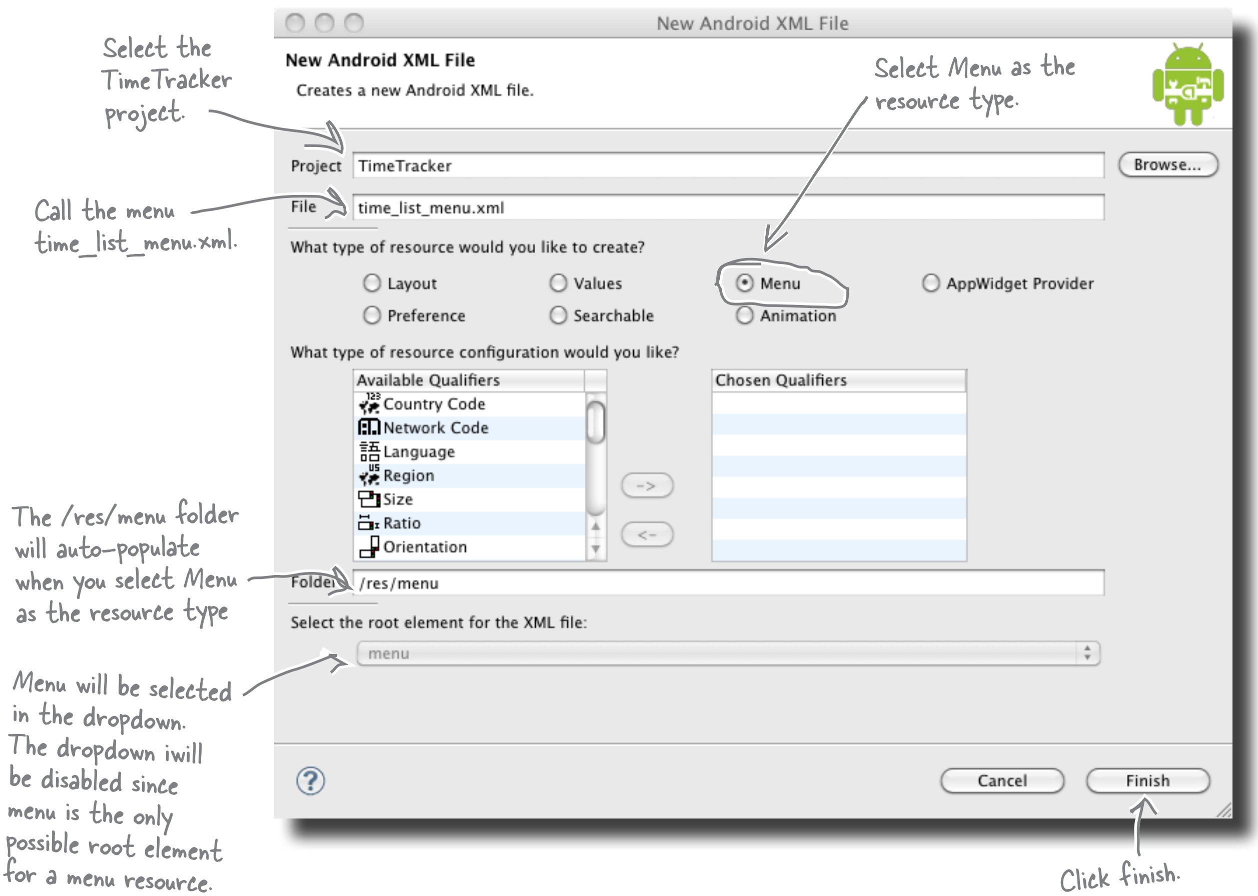
Options menu showing



...but when the menu button is pressed, the menu will show with one button "Add" which will launch the Add Time screen.

Create the menu XML file

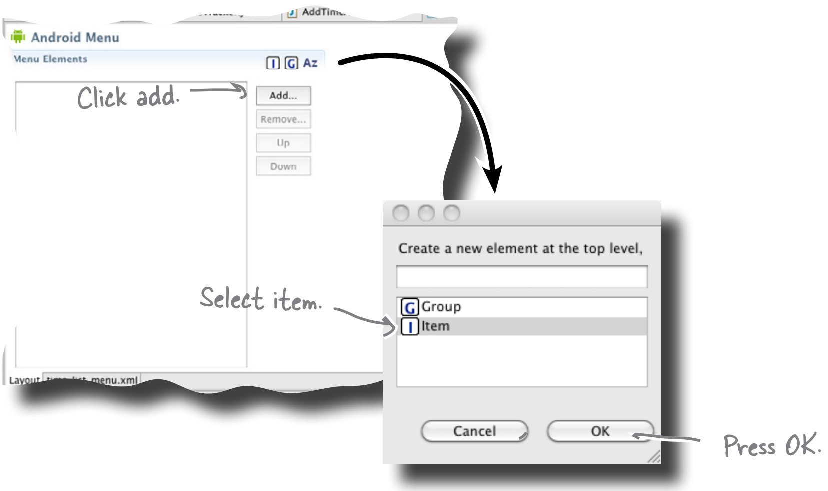
Menus are defined in XML just like layouts and many other Android resources. Just like layouts, you can create new menu XML files using the New Android XML File wizard. Only this time instead of selecting layout options, select menu options.



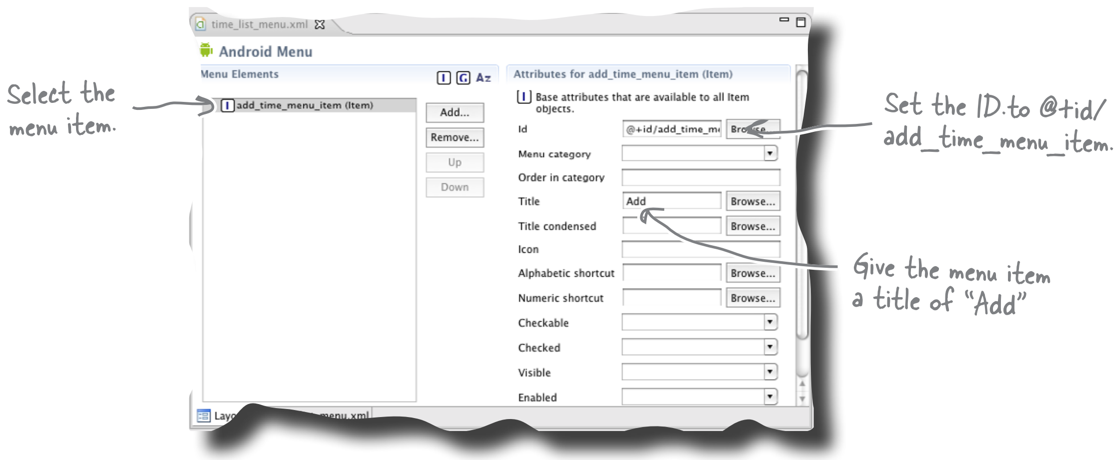
Add a menu option

The menu you just created with the wizard will be in your project under the `res/menu` directory. Navigate to that directory in the Eclipse Package Explorer open it.

Just like the graphical layout editor, there is a graphical editor for creating menus. Start by clicking add to add a new menu item.



Now you can configure the new menu item by setting the title and ID.



Show the menu

Just like XML layouts, the menu is defined in XML, but you need to display it from your Activity. The Activity base class includes a method called `onCreateOptionsMenu` that is called on the displayed Activity when the menu button is pressed. The default implementation does nothing, but you can override it and display your custom menu.

```
public void onCreateOptionsMenu(Menu m) {
        super.onCreateOptionsMenu(m)
        MenuInflater menuInflater = getMenuInflater();
        menuInflater.inflate(
                R.menu.time_list_menu, menu );

}
```

First call super.

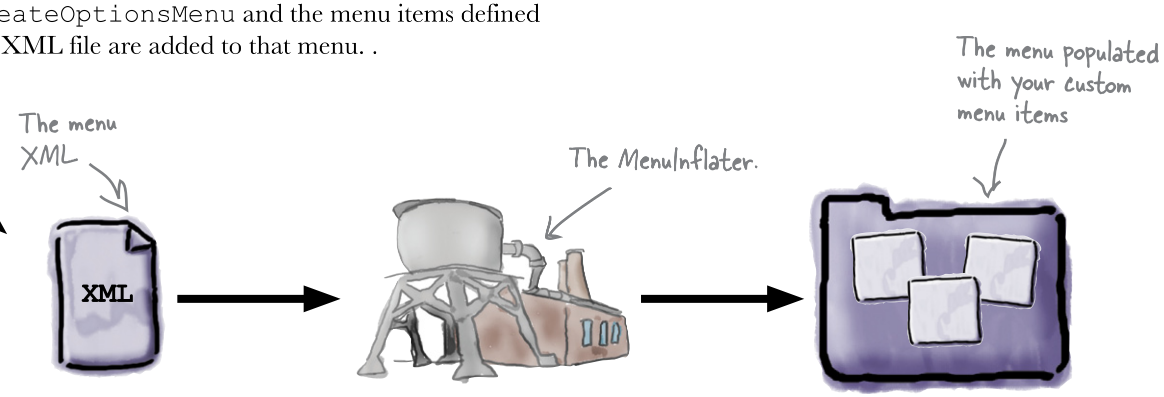
Call Activities method, getMenuInflater() to retrieve the MenuInflater.

Inflate the menu you defined in time_list_menu passing in the R file reference for the menu description.



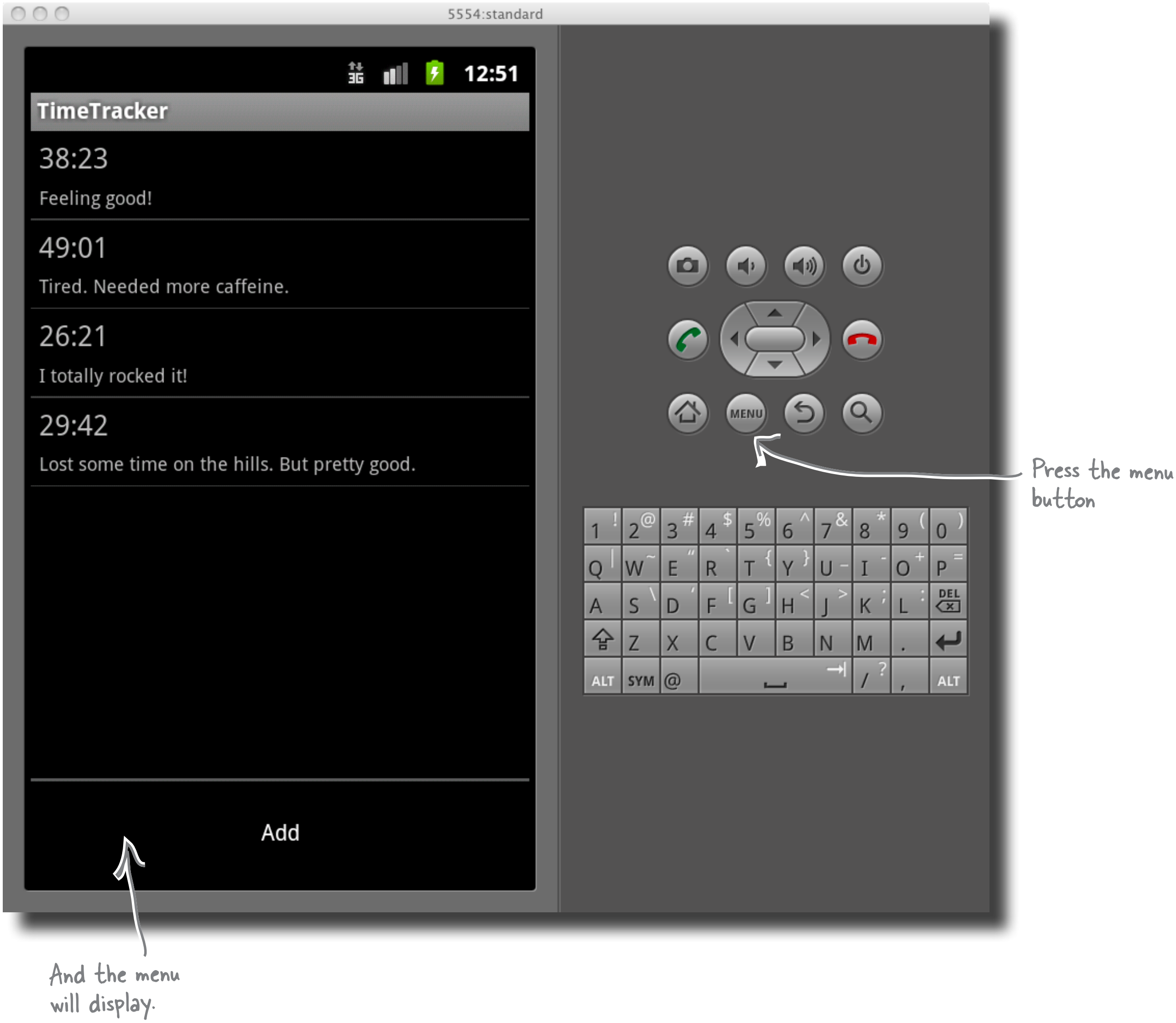
TimeTracker.java

Notice that `onCreateOptionsMenu` uses an Inflater, just like when you inflated the list item layout in the list adapter. The `MenuInflater` takes a menu defined in XML and creates men items. The only difference is that a default menu is passed in to `onCreateOptionsMenu` and the menu items defined in the XML file are added to that menu. .



Test Drive

Run the app, and press the menu button when the time list appears on screen. You should see the menu display with one single item “Add”.



Capture the menu action

There is a companion method to `onCreateOptionsMenu` method called `onMenuItemSelected` which is called when a menu item is selected by the user. To make the menu item work, override `onMenuItemSelected`, check which menu item was selected and invoke your action.

Override onMenuItemSelected

```
public boolean onMenuItemSelected(int featureId, MenuItem item) {
     if (item.id == R.id.add_time_menu_item) {

     }
{
```

Process the add time menu action in here

This method will be called for every menu item you add. It's a good habit to test which item was selected by comparing it to the id you assigned.



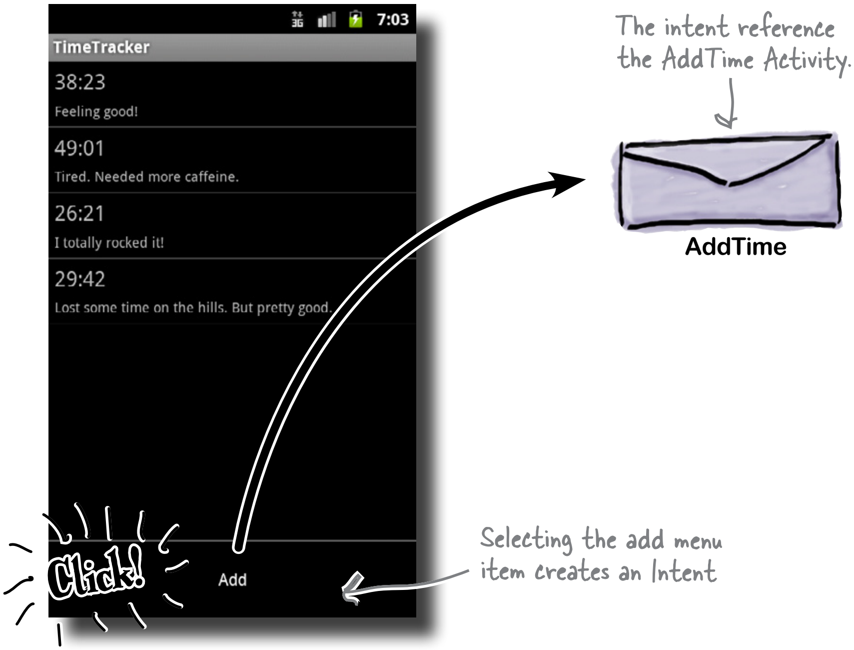
TimeTracker.java

You can add your code to process the menu item inside the if block testing for your menu item. Now you have two independent Activities, and a menu item with an action that can move from one to the other.

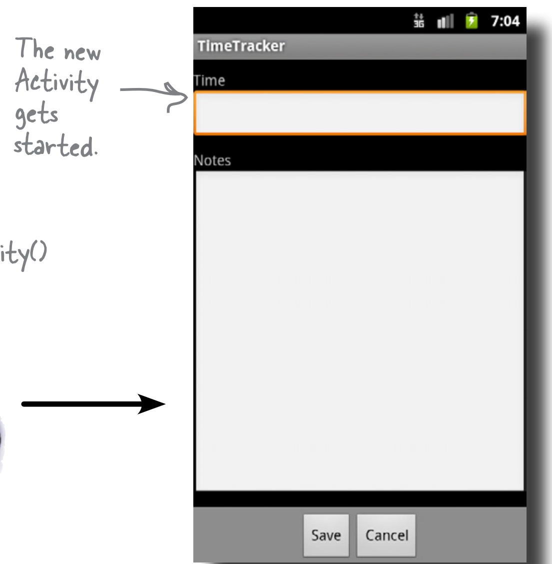
Now turn the page to see how to launch new screens →

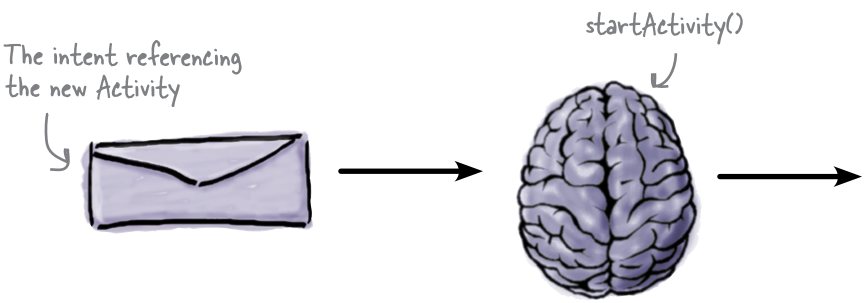
Use Intents to launch new screens

You can launch new screen using an abstract object representation of an action called an **`Intent`**. You can create an Intent when the Add menu item is selected pointing to the `AddTime` Activity.



Then you can call a utility method on the current Activity called `startActivity` passing the Intent. This starts a new Activity in your app, managing all of the lifecycle methods for you including stopping the old Activity as well as creating and startng the new Activity.







Launching a new Activity Magnets

Below is the empty `onMenuItemSelected` method in the `TimeTracker` Activity. Complete the method by creating and invoking an `Intent` to launch the `AddTime` Activity Even though you only have one menu item right now, check and make sure that the ID of the menu item passed in to `onMenuItemSelected` is the add action. Pass the `onMenuItemSelected` call to `super` if you don't process the action.

```
public boolean onMenuItemSelected(int featureId, MenuItem item) {

}
```



TimeTracker.java

```
Intent intent = new Intent(this, AddTimeActivity.class);
```
```
}
```
```
return true;
```
```
if (item.getItemId() == R.id.add_time_menu_item) {
```
```
startActivity(intent)
```
```
return super.onOptionsItemSelected(item);
```

Launching a new Activity Magnets Solution

Below is the `onMenuItemSelected` method in the `TimeTracker` Activity. You should have completed the method by creating and invoking an `Intent` to launch the `AddTime` Activity Even though you only have one menu item right now, you should have checked and made sure that the ID of the menu item passed in to `onMenuItemSelected` is the add action. You should have also passed the `onMenuItemSelected` call to `super` if you don't process the action.

```
public boolean onMenuItemSelected(int featureId, MenuItem item) {

    if (item.getItemId() == R.id.add_time_menu_item) {
        Intent intent = new Intent(this, AddTimeActivity.class);
        startActivity(intent)
        return true;
    }

    return super.onOptionsItemSelected(item);
}
```

Check the item ID to see if the add action was selected

Return true to indicate the select event was processed.

Create and new intent to select AddTimeActivity and then start it

Pass the call on to super for any menu items that may be in the menu.

TimeTracker.java

Open AndroidManifest.xml

Every Activity you use in your app has to be declared in your `AndroidManifest.xml` file. When you created your app with the wizard, it created the Activity for you and added an Activity element in the Android Manifest file.

Before you test the app, add the new Activity declaration to your manifest file or you'll get a nasty exception!

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.headfirstlabs.timetracker"
      android:versionCode="1"
      android:versionName="1.0">
```

The package name for your application

```
      <application android:icon="@drawable/icon" android:label="@string/app_name">
```

The android:name points to the Activity class, by appending the package name to the android:name. So in this case, ".TimeTracker" becomes "com.headfirstlabs.timetracker.TimeTracker"

The label is the text that displays under the icon on the home screen

```
            <activity android:name=".TimeTracker"
                  android:label="@string/app_name">
                  <intent-filter>
                        <action android:name="android.intent.action.MAIN"/>
                        <category android:name="android.intent.category.LAUNCHER"/>
                  </intent-filter>
            </activity>
      </application>
```

This configures the application to be launched from the home screen.

The package name is appended to the android:name, so you just need to enter the class name here.which will give you the fully qualified class name for the AddTimeActivity.

```
      <activity android:name=".AddTimeActivity"> </activity>
```

The activity declaration for the AddTimeActivity.

```
</manifest>
```



AndroidManifest.xml



Test Drive

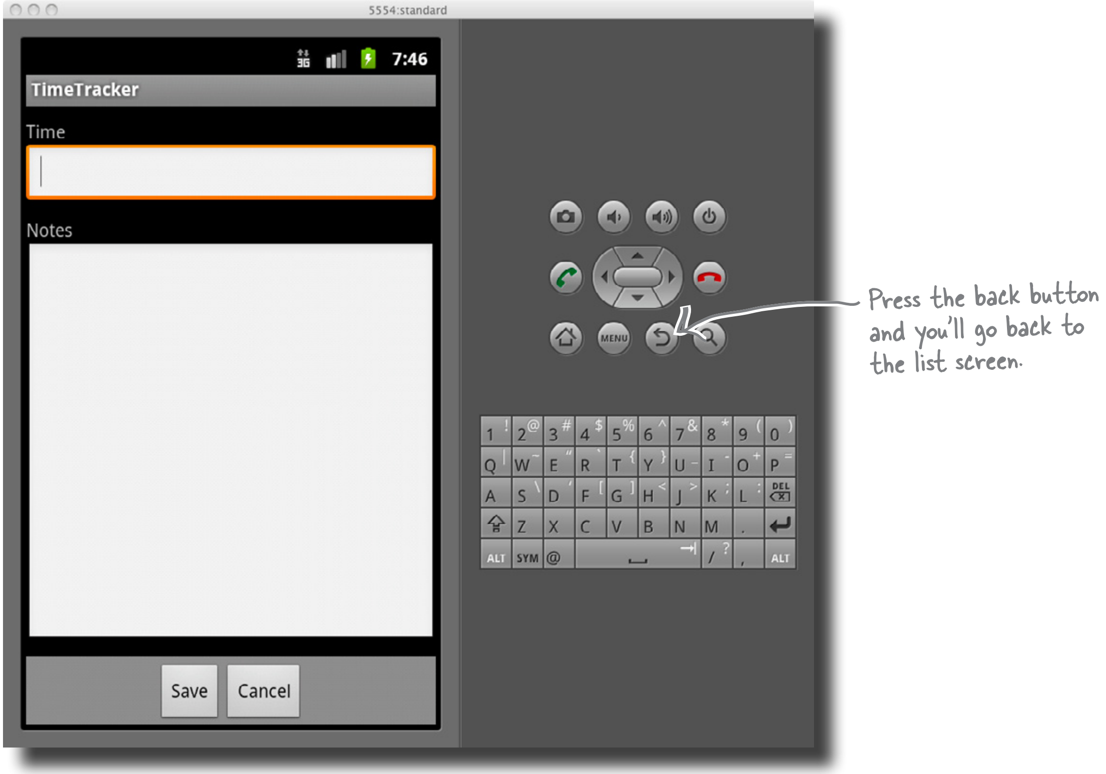
You've got the new screen built, the Intent starting the new Activity from the menu and the new Activity configured in the Manifest. Go ahead and run the app and test out all your hard work!



Perfect! The new screen looks great!

Back Stack Up Close

As you test the app, you'll pretty quickly realize that the save and cancel buttons don't work. But even without implementing these buttons you're not stranded on the new screen. Press the back button and you'll go back to the list screen automatically.



Wait, how did that work?

Android maintains a stack of Activities your app has started, beginning with the first Activity in your app. As you start new Activites like you did with the time entry screen, it's automatically added to the back stack of Activities. And when you press the back button, it automatically goes back to the previous Activity in the stack which in this case is the list screen.

Implement the button actions

The back stack and the back button do allow one way to navigate back to the list screen from the time entry screen, but it's not the behavior you're looking for. You have the **Save** and **Cancel** buttons on screen, and you need to make them work.

Let's start with the **Cancel** button. It layout declaration for the button specifies an `onClick` method called `onCancel`. You could follow the same pattern you used to launch the time entry screen and create a new Intent pointing to the TimeTracker Activity and starting that Intent.

The cancel button's onClick parameter is configured to call a method called onCancel.

```
<Button android:text="Cancel"
    android:onClick="onCancel"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

add_time.xml

```
public void onCancel(View view) {
    Intent = new Intent(this, TimeTracker.class);
    startActivity(intent);
}
```

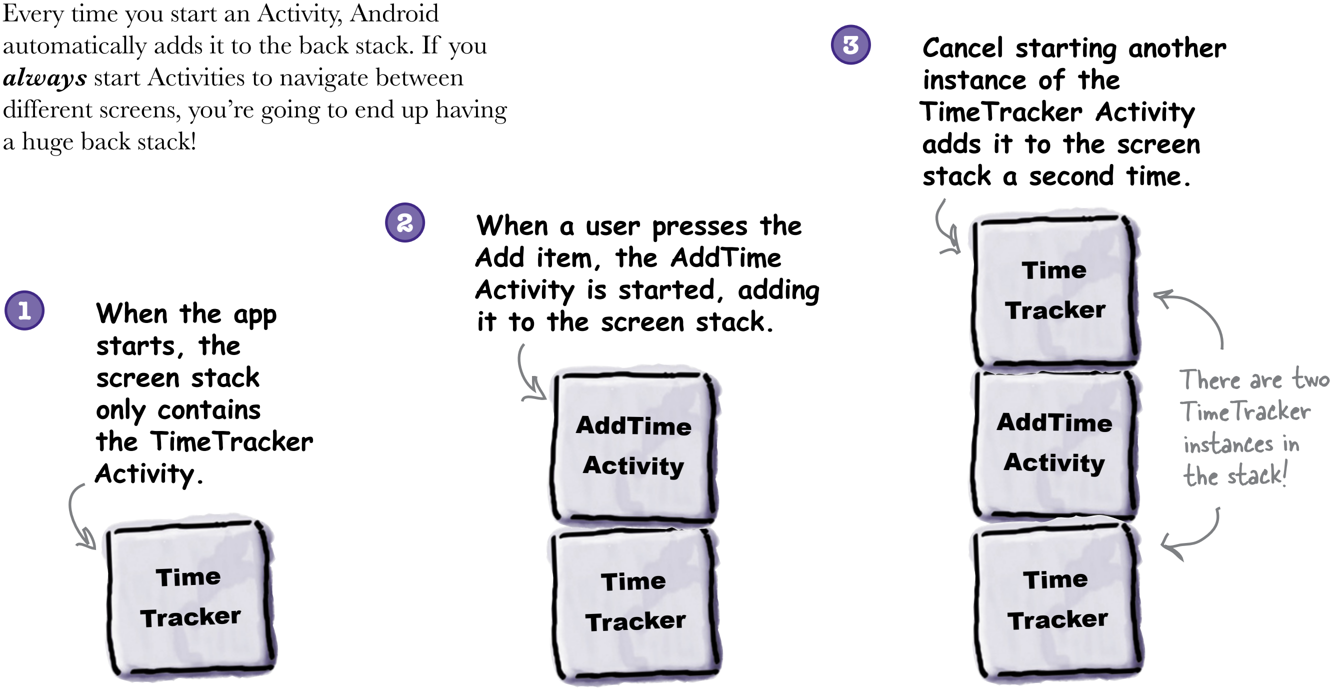
Create an intent to return t the TimeTracker Activity.

Start the Activity.

AddTime.java

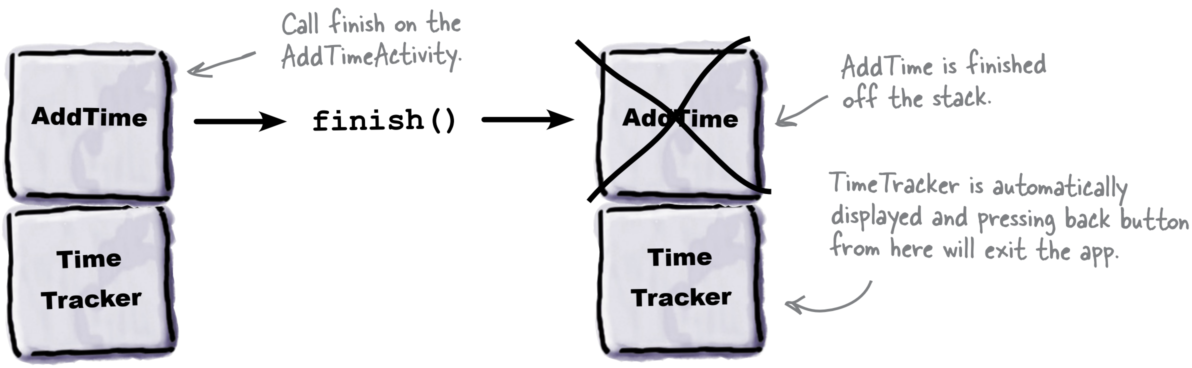
But there's a problem...

Every time you start an Activity, Android automatically adds it to the back stack. If you ***always*** start Activities to navigate between different screens, you're going to end up having a huge back stack!



Take control of the back stack

There are a few different ways to control the back stack. One technique you can use is to call `finish` on the current Activity to end it. This will remove it from the back stack and automatically navigate to the previous screen in the stack.



Implement cancel using finish

If you implement `onCancel` using finish, you'll remove the intent and the `startActivity` call and replace it with a call to finish. This will stop the `AddTime` Activity, remove it from the stack and return the user to the list screen.

```
public void onCancel(View view) {
    Intent intent = new Intent(this, TimeTracker.class);
    startActivity(intent);
    finish();
}
```

Don't start another TimeTracker Activity.

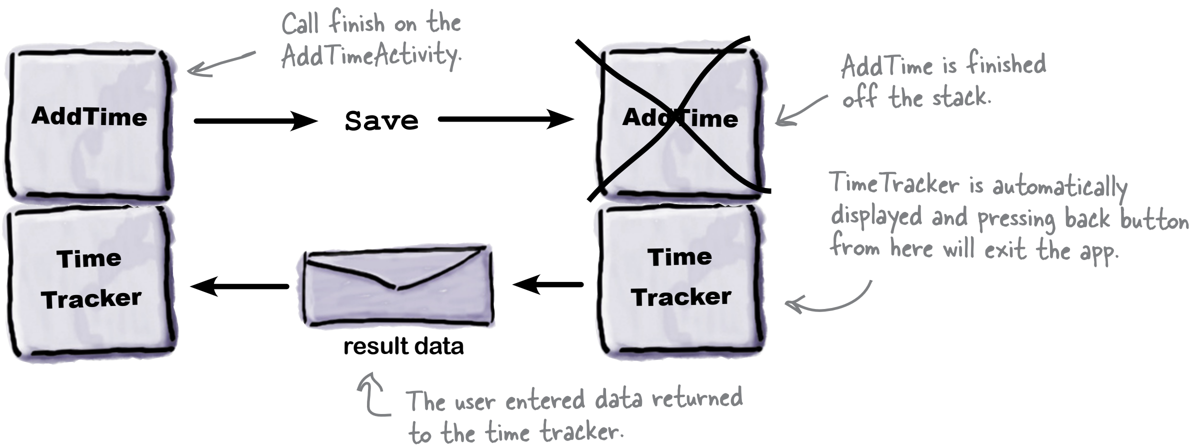
Call finish in the Activity base class.

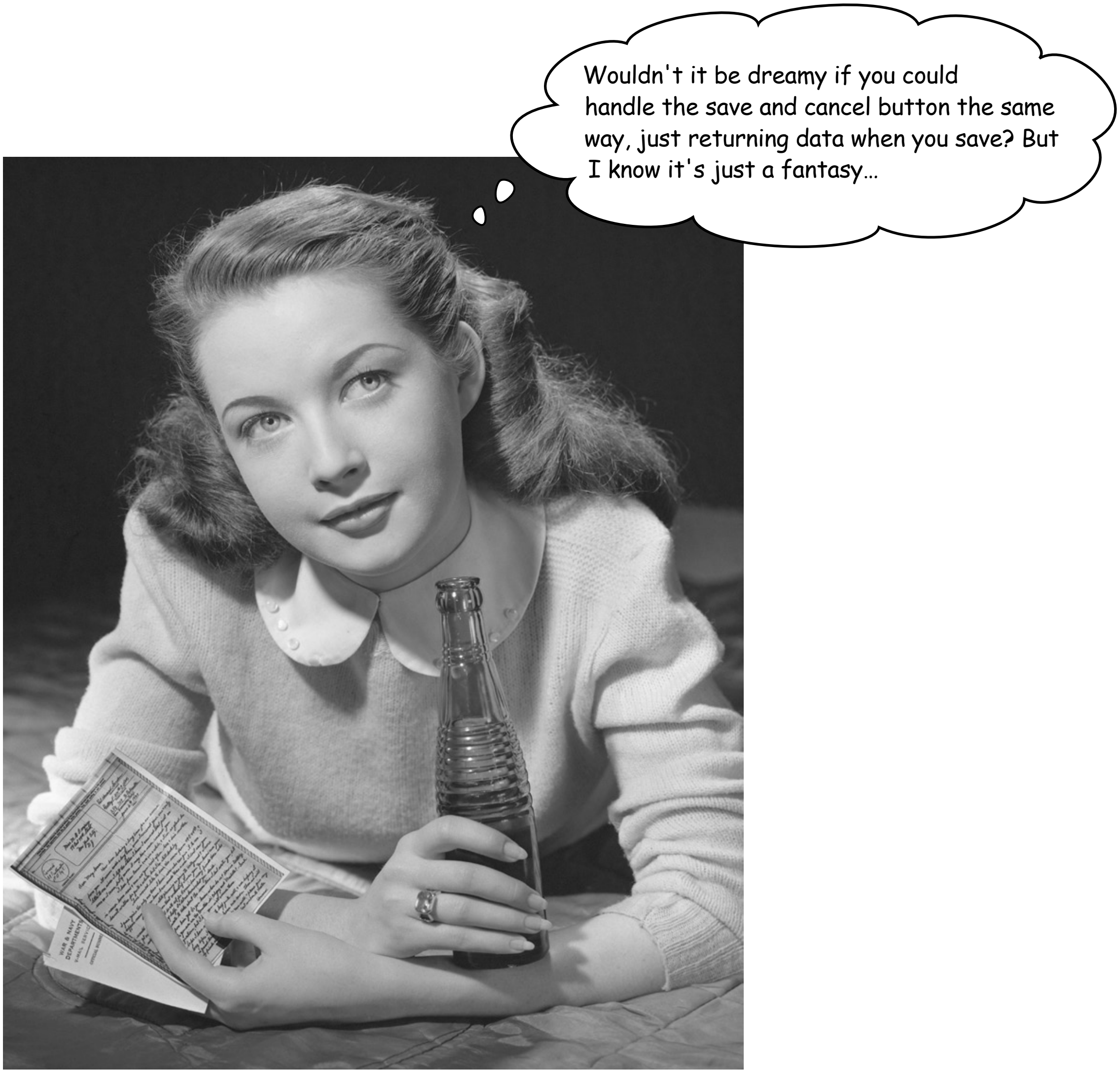


AddTime.java

What about the save button?

This implementation will work for the **Cancel** button, but what about the **Save** button? The Cancel button just needs to return to the list view, but the Save button needs to return to the list view ***and*** return the user entered data.

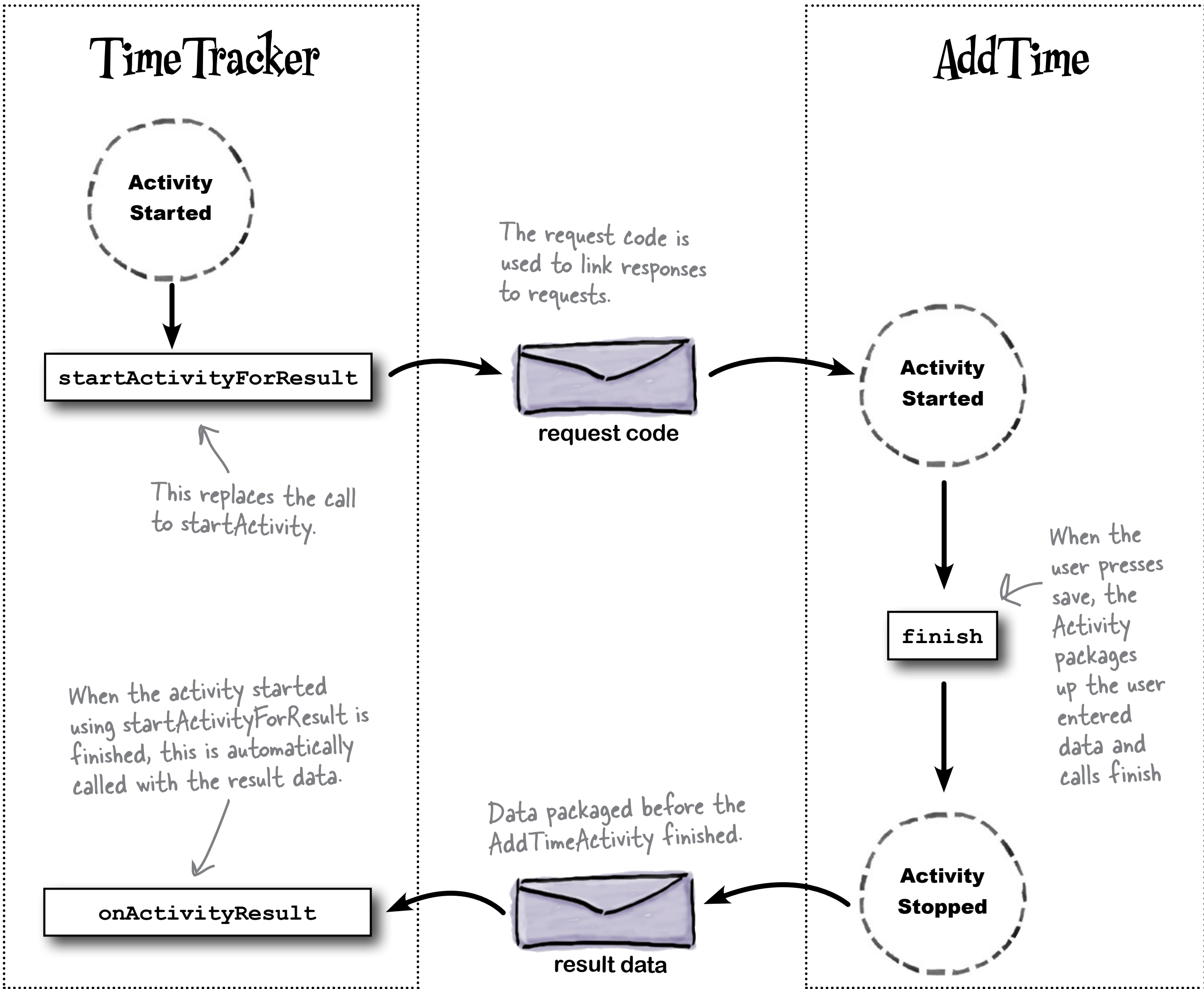



Wouldn't it be dreamy if you could handle the save and cancel button the same way, just returning data when you save? But I know it's just a fantasy...

Use startActivityForResult

There is a mechanism built into Android for launching a new Activity for a result, which is exactly what the TimeTracker is doing by launching the AddTime. The key difference is that the new Activity is started using a special call, `startActivityForResult`. And when the new Activity is finished, a method called `onActivityResult` is invoked on the calling Activity with the resulting data.

Here is the flow between the two Activites



Update starting the Activity

The `startActivityForResult` will work for both the Save and Cancel flows. Before implementing the save functionality, let's go back and update the Save flow to use `startActivityForResult`.

One difference between `startActivity` and `startActivityForResult` is that but you need a request code. This request code is passed back in to the calling Activity when `onActivityResult` is called so the you can correlate the responses to the screens you've started.

```
public static final int TIME_ENTRY_REQUEST_CODE = 1;
```

The request code constant.

Now remove the `startActivity` call and instead call `startActivityForResult` passing in the intent and the request code.

```
public boolean onMenuItemSelected(int featureId, MenuItem item) {
      if (item.getItemId() == R.id.add_time_menu_item) {
             Intent intent = new Intent(this, AddTimeActivity.class);
             startActivity(intent);
             startActivityForResult(intent, TIME_ENTRY_REQUEST_CODE);

             return true;
      }

      return super.onOptionsItemSelected(item);
}
```

Replace the startActivity call with a call to startActivityForResult.

Pass in the time entry request code constant.



TimeTracker.java

Implement onSave

The **Cancel** flow looks great, so let's move on to the **Save** flow. You'll start by implementing the `onSave` method invoked by the Save button when clicked. You'll implement this method in the `AddTime` Activity.

```
<Button android:text="Save"
      android:onClick="onSave"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content" />
```



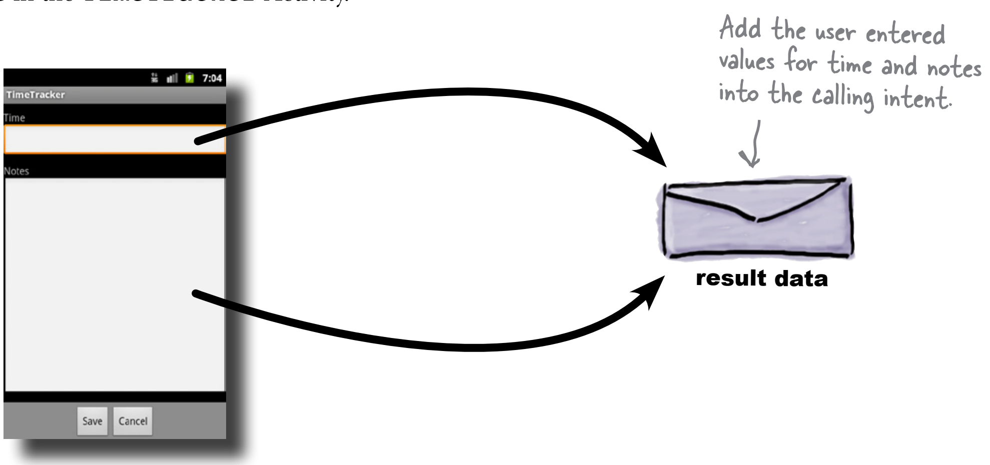
add_time.xml

```
public void onSave(View view) {
      ← Your implementation here →
}
```



AddTime.java

In the `onSave` method, you'll retrieve the `EditTexts` for the time and notes fields from the view. The `Intent` that invoked the `AddTime` is going to be returned to the `TimeTracker` Activity. So you can put these values in a `Map` inside the `Intent`. Then you can retrieve those values from the `Intent` in the `TimeTracker` Activity.





onSave Magnets

Below is the empty `onSave` method from the `AddTime` Activity. Use the magnets below to complete the method. You'll need to retrieve reference to both `EditTexts` as well as the Intent. Then use Intent's `putExtra` method to add values to the Intent's Map so that you can retrieve them later from the `TimeTracker` Activity. Finally set the result of the Intent to `RESULT_OK` which you'll use in the `onActivityResult` method to determine whether the Save or Cancel button was pressed. .

```
public void onSave(View view) {

}
```



AddTime.java

```
intent.putExtra("notes", notesView.getText().toString());
```

```
EditText notesView = (EditText)findViewById(R.id.notes_view);
```

```
intent.putExtra("notes", timeView.getText().toString());
```

```
EditText timeView = (EditText)findViewById(R.id.time_view);
```

```
this.setResult(RESULT_OK, intent);
```

```
finish();
```

```
Intent intent = getIntent();
```



onSave Magnets Solution

Below is the `onSave` method from the `AddTime` Activity. You should have used the magnets below to complete the method. You should have retrieved references to both `EditTexts` as well as the Intent. Then using the Intent's `putExtra` method, you should have added values to the Intent's Map so that you can retrieve them later from the `TimeTracker` Activity. Finally you should have set the result of the Intent to `RESULT_OK` which you'll use in the `onActivityResult` method to determine whether the Save or Cancel button was pressed.

```
public void onSave(View view) {

    Intent intent = getIntent();

    EditText timeView = (EditText)findViewById(R.id.time_view);
    intent.putExtra("time", timeView.getText().toString());

    EditText notesView = (EditText)findViewById(R.id.notes_view);
    intent.putExtra("notes", notesView.getText().toString());

    this.setResult(RESULT_OK, intent);
    finish();

}
```

Calling getIntent() retrieves the starting intent from a running Activity.

Get a reference to the time EditText, and put its value in the intent using the string constant.

Get a reference to the notes EditText, and put its value in the intent using the string constant.

Set the result to OK and pass in the intent.

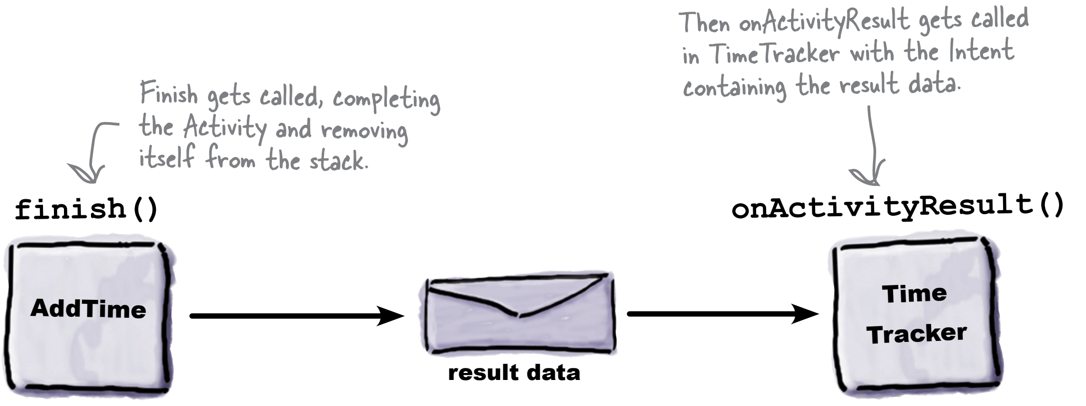
Finish the activity.



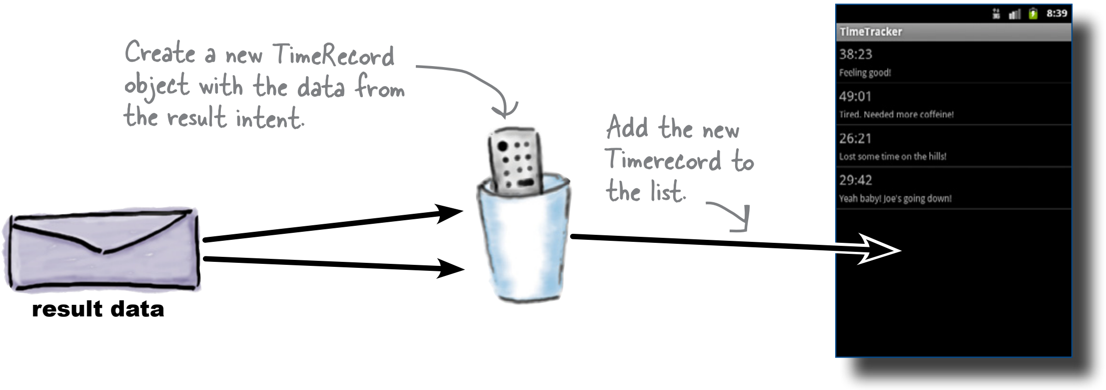
AddTime.java

Implementing onActivityResult

You've completed the `onSave` method, which packages up the user entered data in the calling intent. It also calls finish on its Activity which pops that Activity off the stack and returns to the `TimeTracker` Activity, calling its `onActivityResult` method.



In the `TimeTracker` `onActivityResult` method, you'll retrieve the values from the Activity using the `getStringExtra` method, using the map keys used to add the values. Then you'll create a new `TimeRecord` object with the values and add it to the `ListAdapter`.



Pool Puzzle



Your **job** is to take the code fragments from the pool and place them into the `onActivityResult` method. You may **not** use the same code fragment more than once. Your **goal** is to make a new item display in the list..

```
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
        if (requestCode == TIME_ENTRY_REQUEST_CODE) {
                if (resultCode == RESULT_OK) {







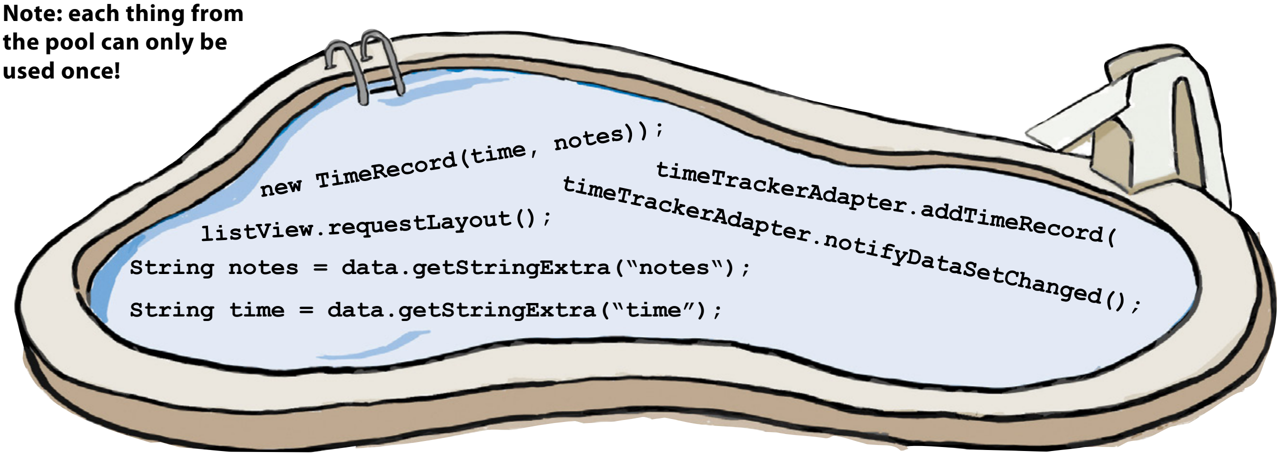


                }
        }
}
```

This check makes sure the requestCode is the code you passed in

This checks that the resultCode is RESULT_OK. Since you didn't set the result code in the onCancel, this will return instead of trying to add a new item.

Note: each thing from the pool can only be used once!





Intents Exposed

This week's interview:
Are Intents Under Appreciated?

Head First: Hi Intent, thanks for speaking with us tonight.

Intent: Happy to be here, try and tell my story a little bit, you know.

Head First: Wow, your story? Sounds like you have something on your mind. What's up?

Intent: It's nothing new really. I just don't get a lot of respect around here. I mean, I can do an awful lot! I help start Activities, I let everyone know where to go, and I can store and communicate data myself as I move around the system.

Head First: That all sounds right. But it sounds like you're not too happy about it.

Intent: I feel bad coming here and complaining, but I just never get to see the spotlight you know? Activities get to interact with users! I just have to hang out in the background while they get to shine on the screen.

Head First: It must be awful for you to just sit there while the Activities are out there displaying themselves to users, getting their buttons pressed...

Intent: Hey! You don't have to rub my face in it, Okay?

Head First: Oh, I'm sorry, I didn't mean...

Intent: It's Okay. I'm used to it.

Head First: No, I'm telling you that you **are** really important. You may be sitting in the background while the Activity is displayed, but you have to keep track of ***really important information***.You know how the Activity was launched, and you include any information passed in to the Activity.

Intent: That's true...

Head First: And as you're sitting there in the background while the Activity is displaying, you get asked for your information and new information gets passed to you. Like when information is added to you to get sent back to a calling Activity after calling startActivityForResult.

Intent: That's true too.

Head First: I think you need to change your mindset. You're not under appreciated, you're the ***strong silent type***.

Intent: The strong silent type... I think I like the sound of that.

Head First: Glad you're feeling a bit better. That's all the time we have tonight folks. Give Intent a big round of applause before going back into the background and we forget about it!

Intent: Hey now!

Head First: Kidding, man. Kidding.

Pool Puzzle Solution



Your **job** is to take the code fragments from the pool and place them into the `onActivityResult` method. You may **not** use the same code fragment more than once. Your **goal** is to make a new item display in the list.

```
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
        if (requestCode == TIME_ENTRY_REQUEST_CODE) {
                if (resultCode == RESULT_OK) {
                        String notes = data.getStringExtra("notes");
                        String time = data.getStringExtra("time");

                        timeTrackerAdapter.addTimeRecord( new TimeRecord(time, notes));

                        timeTrackerAdapter.notifyDataSetChanged();

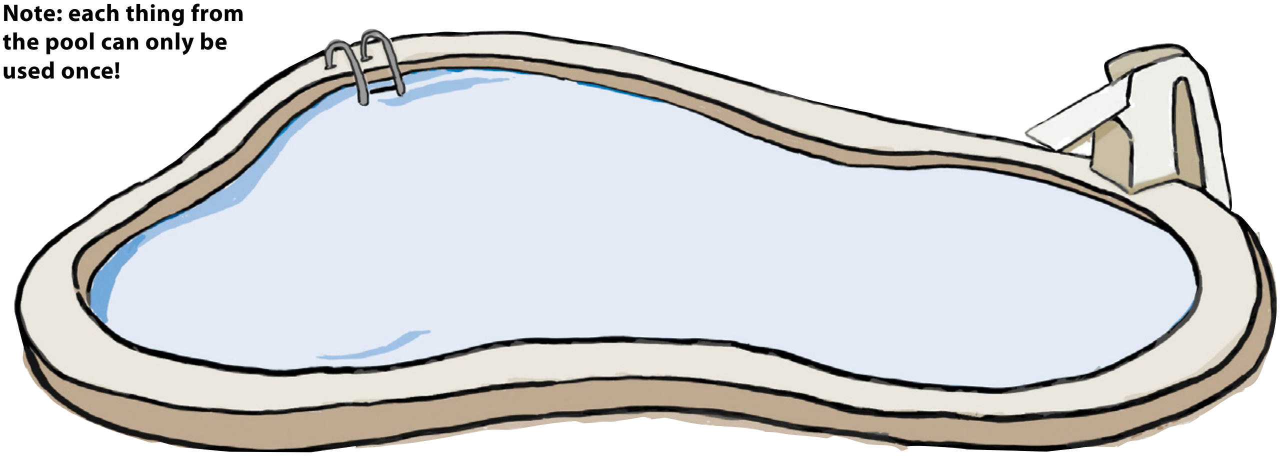
                }
        }
}
```

Get the values from the intent (the two `getStringExtra` lines).

Create a new TimeRecord and add it to the list adapter (the `addTimeRecord` line).

This method lets the list know the data has changed and to update the display (`notifyDataSetChanged`).

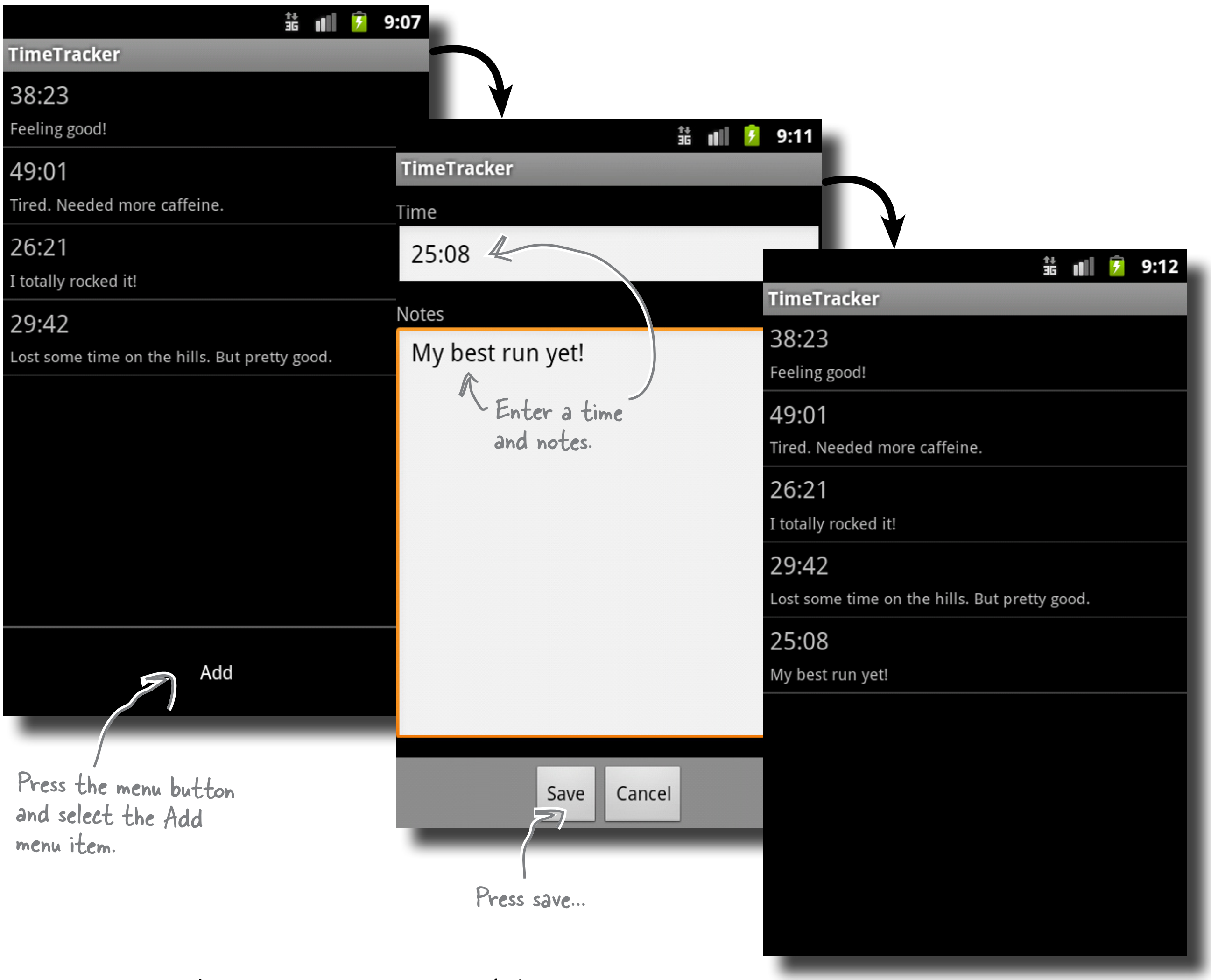
Note: each thing from the pool can only be used once!





Test Drive

Everything is all wired up! Run the app and run through the complete flow of adding a new time. Invoke the Add menu item, enter a time and some notes, and press save. And you'll see a new item added to the list!



Fantastic Work!



Go Off Piste

You just did some seriously heavy lifting to get data entry working. Can't get enough? Here are some more features you could implement to make the app even better!

Build edit and delete

In this chapter, you built a mechanism to add items to the list.. But what if a user enters the wrong information? Allowing users to add information is great, but your users will eventually want to be able to edit and delete as well.

Build an about screen

The bulk of the navigation in this chapter used startActivityForResult to manage data entry. Try building another screen, like an about screen, that displays but doesn't return data to the calling Activity. Think about whether you want that Activity to be in the back stack and build it accordingly.



Your Android Toolbox

Now that you've built navigation between two screens, you can apply the same logic to building navigation between as many screens as you like! Just not too many, OK?

Screen Navigation

- Create a new Activity and configure it to use a new Layout
- Create an Intent
- Call startActivity or startActivityForResult to launch a new screen

New Menu Steps

- Create a menu XML file from the new XML file wizard
- Add menu items using the graphical editor, or edit the raw XML.
- Inflate the menu using the MenuInflater in the onCreateOptionsMenu method in your Activity
- Process the menu action in onMenuItemSelected in your Activity.

BULLET POINTS

- Create new Layouts using the new XML file wizard, or by creating the XML files yourself.
- Reuse Activities with different layouts if the behavior is the same. If the behavior is different, create a new Activity.
- Remember to add a declaration for your new Activity in AndroidManifest.xml. If you don't you'll get nasty errors!
- To launch a new Activity in your app, create an Intent and pass it to `startActivity`.
- If you're staring an entry screen, use `startActivityForResult` to easily finish and return values to the calling Activity.
- Implement `onActivityResult` to receive the data returned from the screen.
- Create new Context Menu XML descriptions using the new XML file wizard.
- Show menus by overriding Activities `onCreateOptionsMenu` and process the selection events by overriding `onMenuItemSelected`.
- New screens are automatically added to the back stack. The back buttons uses this back stack when pressed.
- Call `finish` to complete a screen and automatically display the previous screen on the back stack.
- Use EditText for text entry

9 database persistence

Store your stuff



In memory data storage only gets you so far. In the previous chapter, you built a list adapter that only stored data in memory. But if you want the app to ***remember*** data between sessions, you need to **persist** the data. In this chapter, you'll learn to store your data using a SQLite database. You'll learn how to create and manage your own SQLite database and you'll learn how to integrate that SQLite database with the ListView in the TimeTracker app. And don't worry, even if you're brand new to SQL, you'll learn what you need to get this app's database up and running.

Uh oh, the times aren't saving...

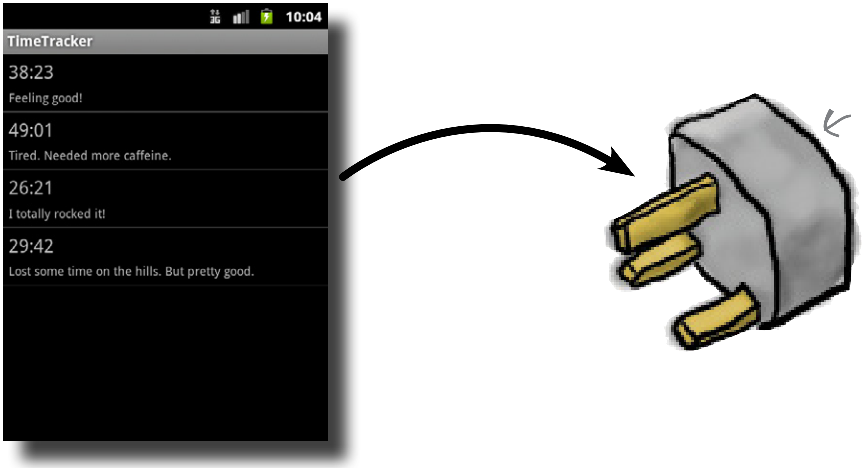
Donna is loving the app so far. It's a straightforward app where she can enter her times and notes. And just like she wanted, it's free of clutter from features she won't use.

But she pointed out a ***really big problem***. When she closed the app and later reopened it, **all of her times were gone**!

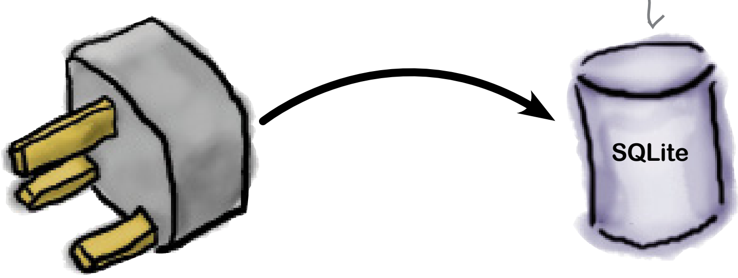


... but you can save them using SQLite

The app currently loses all of the information added to list when you exit and relaunch the app. This is because newly entered times are stored in memory as objects inside the `TimeTrackerAdapter`. And once you shut down the app, the in memory data is gone!



Android comes standard with a built in SQLite database implementation. SQLite is a lightweight SQL database implementation that stores data to a text file on the device. If you store the times in the SQLite Database and read them back in after you restart the app, you'll have persistent data.

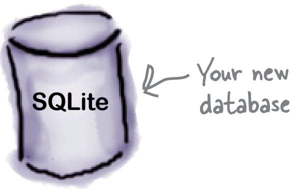


Storing times in the database

You'll have to touch several parts of the app to get database storage fully integrated. Let's take a look at what you'll be doing in the chapter to seamlessly persist data.

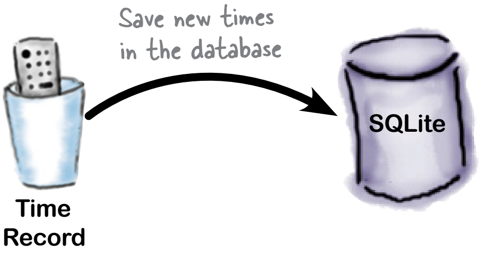
1. Create a database for your app

You'll be storing the time and note data in a SQLite database. But before you can store data in the database, you have to create it.



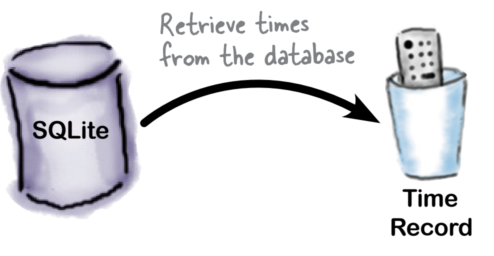
2. Save a time record

Once the database is created, you can save times in it. Here you'll define the database schema based on the data you'll be saving. Then add the code to insert records directly into the database.



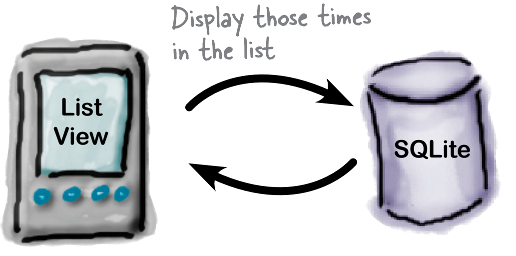
3. Load time records

It's no fun to store data if you can't access it. Here you'll write the code to *query* the database and process the results.



4. Update the List to use the database

The goal is not to save and load data from a database in isolation. The goal is integrate database persistence in the existing app. With store and retrieval working, you'll finish up by integrating all of your hard work back into the TimeTracker app.



Start by creating the database

You can create and open databases directly inside your app. The best way to get off the ground with a new database is to extend a built in abstract base class called `SQLiteOpenHelper` that provides you with all of the basic behavior to manage a database.

Create a new class called `TimeTrackerOpenHelper` that extends SQLiteOpenHelper. There are three methods you'll need to implement that descrive how to connect to your database, initially create tables, and upgrade from previous versions.

Do this!

> Create a new class called `TimeTrackerOpenHelper` that extends SQLiteOpenHelper. Pass the database name and the database version to super. Make empty implementations of `onCreate` and `onUpgrade`.

```
private static class TimeTrackerOpenHelper extends SQLiteOpenHelper {

    TimeTrackerOpenHelper(Context context) {
        super(context, "timetracker.db", null, 1);
    }
```

Pass the name of the database to super

Pass the version number to super as well.

```
    public void onCreate(SQLiteDatabase database) {
```

Create your tables in here

```
    }

    public void onUpgrade(SQLiteDatabase database,
        int oldVersion, int newVersion) {
```

Handle database schema upgrades in here

```
    }
}
```



TimeTracker OpenHelper.java

Instantiate the OpenHelper

The database is created internally by the Open Helper when it is instantiated. In `TimeTracker`, add the following line creating an instance of the `TimeTrackerOpenHelper`.

Do this!

Add the line to instantiate the `TimeTrackerOpenHelper` in the `TimeTracker` `onCreate` method, then start the app.

```
public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        ListView listView = (ListView)findViewById(R.id.times_list);
        timeTrackerAdapter = new TimeTrackerAdapter();
        listView.setAdapter(timeTrackerAdapter);

        TimeTrackerOpenHelper openHelper = new TimeTrackerOpenHelper(this);

}
```

Instantiating your custom open helper will cause the database to be created.



TimeTracker.java

there are no Dumb Questions

Q: Do I have to call a method on the OpenHelper to create the database?

A: No. When you instantiate the OpenHelper, it automatically creates the database for you.

Q: Cool! Where does it go?

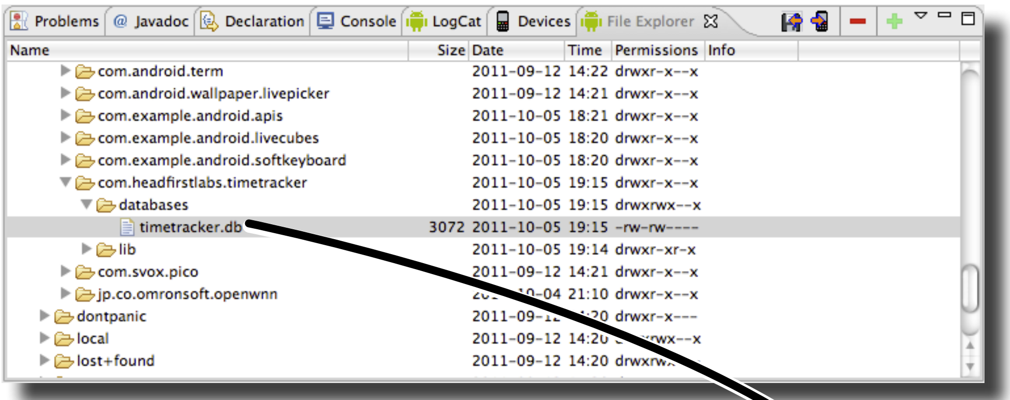
A: It's stored on the device under /data/data/<package-name>/ databases<database-name>. If you're ever curious about what's in the database, you can always open it up in SQLite databse browser and look at its contents.

Browse to the database file

After running the app with the Open Helper being created, you won't notice any visual differences. But there are **big changes behind the scenes**. When you instantiated the Open Helper, the database file was created and saved to your applications persistent storage.

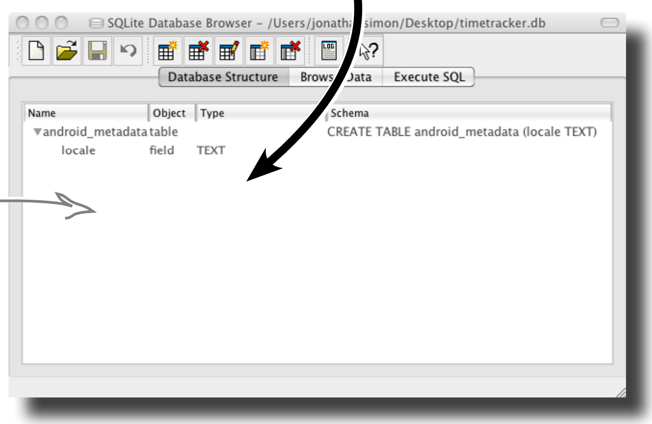
You can view the file by opening the Android File Explorer. Go to Window → Show View - Other, expand the Android folder and select File Explorer. Then navigate to `com.headfirstlabs.timetracker\databases\` and you'll see a file called `timetracker.db`.

The save button to save the database to your file system.



Select the database file and press the save icon. This will allow you to save the entire database file locally and view it. Here is a screenshot of the **sqlitebrowser** (http://sourceforge.net/projects/sqlitebrowser/) displaying the contents of the database. Right now the database is empty, it just includes some default metadata.

The sqlitebrowser viewing the sqlite database file.



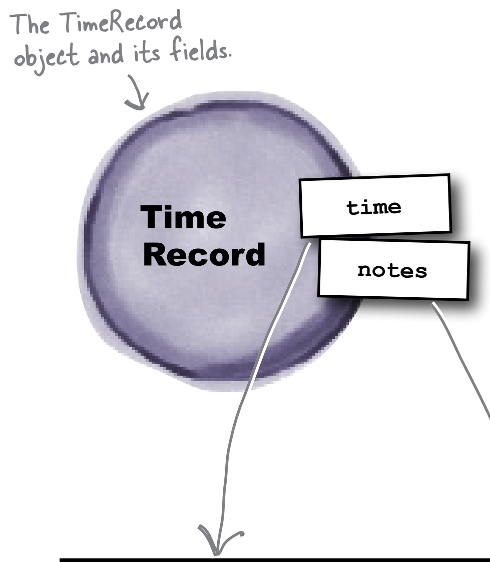
Design the database

You now have the database being created with the open helper. *But it's empty.* Now look at what you need to store and how to structure the database to store that information. The data for this app are already stores in the `TimeTrackerAdapter` in a list of `TimeRecord` objects. Now you need to store that same information in the database.

You can store this by creating a single table called `timerecords` with a column for time and notes.

Geek Bits

Notice that the sqlite database file is called timerecords.db. The file's name is controlled by the database name string you passed into the constructor of the SQLiteOpenHelper.



id	time	notes
1	38:23	Feeling good!
2	49:01	Tired. Needed more caffeine!
3	26:21	I totally rocked it!
4	29:42	Lost some time on this hills. But pretty good.

The ID field is the primary key for the database. This is standard practice for databases.

Sample data

Create the initial table

The database design includes the one `timerecords` table that you'll need to create when the database is created. You overrode the `onCreate` method in `SQLiteOpenHelper` when you wrote the `TimeTrackerOpenHelper` which created a blank database. Now that you know what the database should look like, you need to include the code to create that creates the initial table. Here is the SQL you'll need to execute.

SQL statement to create the timerecords table

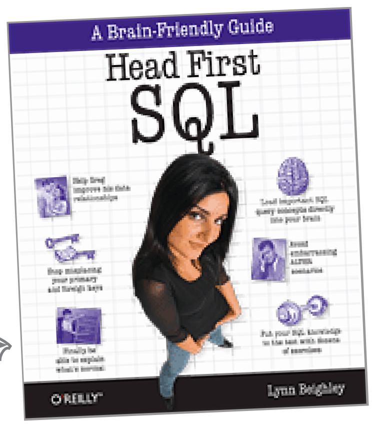
```
create table timerecords (
     id integer primary key time text, notes text
)
```

there are no Dumb Questions

Q: How much SQL do I need to know for developing Android apps?

A: That really depends on your app. Some apps just set up a very basic database and display its contents. Others do very complex things with their database, like very detailed queries using very intricate database schemas. We won't go into a lot of detail about the SQL part of SQLite in this book. If you'd like to know more, we can suggest you read Head First SQL.

Our very biased suggestion on where to learn more about SQL.



Updating the database creation

Update your `onCreate` method to the following. A `SQLiteDatabase` instance is passed in which is an Object wrapper around the SQLite Database. You can execute SQL using the `execSQL` method.

```
public void onCreate(SQLiteDatabase database) {
    database.execSQL(
        "create table timerecords " +
        "(id integer primary key, time text, notes text)"
    );
}
```

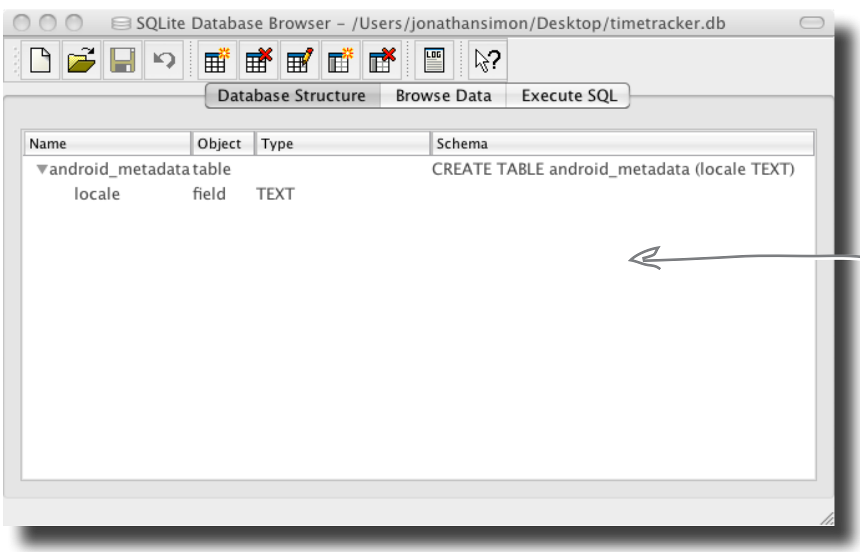
Call execSQL on the database

Pass in the SQL statament to create the timerecords table.



TimeTracker OpenHelper.java

If you run the app again, you still won't see any visual or functional change in the app. But you did update the `TimeTrackerOpenHelper onCreate` to update the database creation. So check the sqlite database file for schema changes.



Looks the same as before. No new table in the database structure.



It's because the database is *persistent*

The SQLiteOpenHelper is helper class for creating and managing the SQLite database, which you've seen is stored in a file for persistence. This way, data stored in the file will be available after the app process is exited and restarted.

But the code that was just updated was for `onCreate` which only gets called when the database is created. The database doesn't get created each time your app runs though, only the very first time. That's what makes the data persistent.

Keep reading to see how to update the database →

Implement onUpgrade

At this point you have a database you need to update. You need to add the `timerecords` table to the original empty database. This pattern of updating a database's schema is common so the open helper provides a mechanism for it.

In the `TimeTrackerOpenHelper` constructor, you passed a version number of the database to super which is cached along with the database. If the version number changes, `onUpgrade` is called for you to update the database as needed.

In this case, the upgrade will be quite simple. You just need to drop the database and recreate it.

```
public class TimeTrackerOpenHelper extends SQLiteOpenHelper {

        TimeTrackerOpenHelper(Context context) {
                super(context, "timetracker.db", null, 2);
        }

        public void onCreate(SQLiteDatabase database) {
                database.execSQL(
                        "CREATE TABLE timerecords " +
                        "(id INTEGER PRIMARY KEY, time TEXT, notes TEXT)"
                );
        }

        public void onUpgrade(SQLiteDatabase database, int oldVersion, int newVersion) {
                database.execSQL("DROP TABLE IF EXISTS timerecords");
                onCreate(database);
        }
}
```

Update the version number passed in to super (pointing to `2`)

Drop the tables if they exist and then call onCreate. (pointing to the two lines in `onUpgrade`)



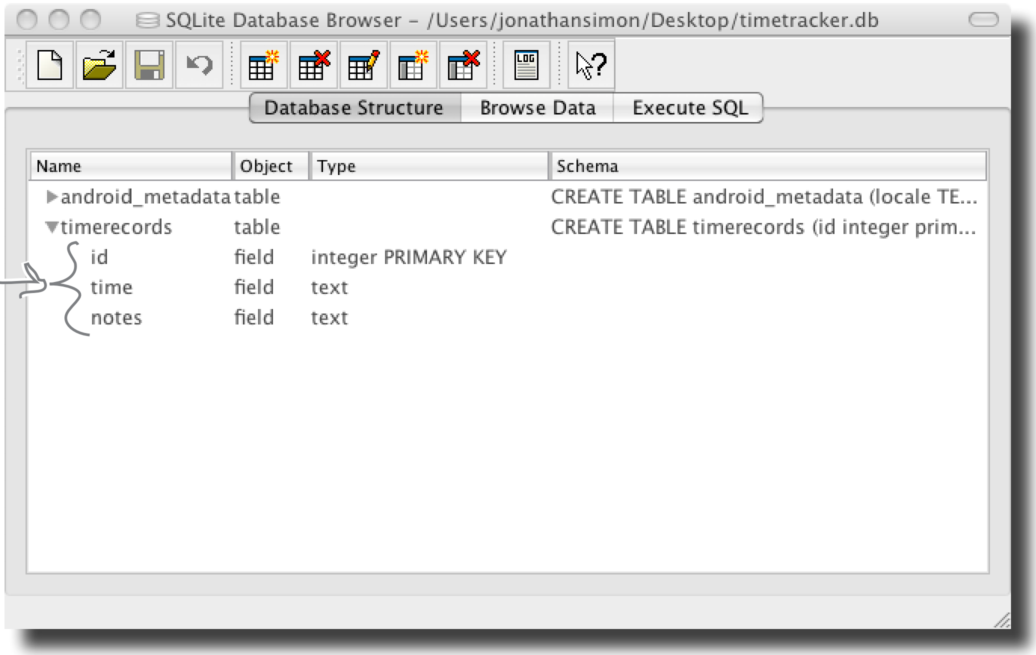
TimeTracker OpenHelper.java



Test Drive

Now that you've updated `onCreate`, updated the version number and implemented the `onUpgrade` method, it's time to test this out. Run the app again and inspect the sqlite file in a viewer.

Here is the new table and fields.



The database is updated!



Don't forget to update the version number.

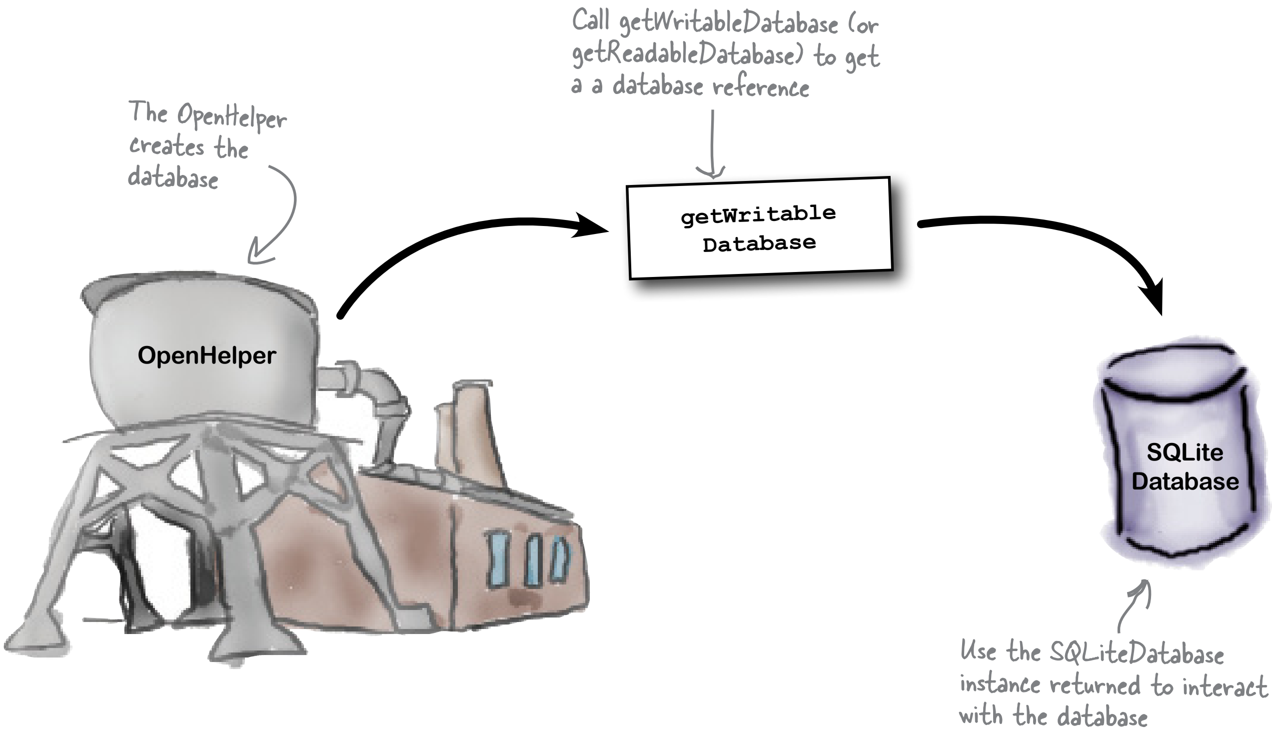
The onUpdade method will only get called if the version number.is updated. If you update your database schema, make sure to update the version number or the database will not get updated to the latest version.

Using the database in your app

The `OpenHelper` isn't a database itself. But it does create the database for you, and gives you access to it. You don't have to manually create the database, that's done for you when you instantiate the `OpenHelper`. But you do need to call one of the `getDatabase` method to retrieve a reference to the `SQLiteDatabase` object.

Once you have the SQLiteDatabase, you can call any of the methods to insert, delete, execute raw SQL statements, and more. But first, you need to get a reference to the database from the `OpenHelper`.

There are two methods you can call to retrieve the database, `getReadableDatabase` to retireve a read only database and `getWritableDatabase` and to retrieve a database you can read and write to. Since you'll be writing to the database when you add new times, you'll be calling `getWritableDatabase`.





SQLiteDatabase Exposed

This week's interview:
What are you, exactly?

Head First: SQLiteDatabase, thanks for joining us. I know it's hard to time away from your server to join us here tonight.

SQLiteDatabase: Thanks! But you know, I don't have a **server**, that's just ***soooo*** old school. I'm an *individual*. I work alone. I refuse to be downtrodden by the shackles of a server...

Head First: Wow! OK, so no server. Gotcha. What exactly do you need to run?

SQLiteDatabase: Sorry, I get a little carried away sometimes. My whole point is to run *minimally*. You can just drop my library anywhere, and without any configuration, setup, additional processes or weird data storage, you have a perfectly functional SQL database.

Head First: Seriously? If you don't have your own process, where do you run?

SQLiteDatabase: I'm pretty flexible, you know. I run in whatever process runs my library. I run the their process. But I don't take much. I'm a drifter.

Head First: Cool! And where do you store your data?

SQLiteDatabase: On the regular file system in a plain old file.

Head First: Between running as a configureless library and storing your data in a plain file, is your functionality limited?

SQLiteDatabase: No way! I'm super powerful. I can do multiple tables, triggers, indeces and all kinds of fancy stuff like that in my one little file.

Head First: Wow, I'm impressed!

SQLiteDatabase: You should be. Also, I weight a pretty slim 350k. But when apps need me to be super small, I have a special diet I can go on and drop down to under 200k. I'm just cool like that.

Head First: Stop, you're killing me! How do you fit that all in there?

SQLiteDatabase: A lot of folks use me, and they care a lot about making sure I'm super optimized. I have my own **consortium**, you know.

Head First: Seriously?

SQLiteDatabase: Yeah! You should check it out, sqlite.org. You can see all of the folks there that make me happen.

Head First: That's amazing! Tell me a bit about your object representation on Android.

SQLiteDatabase: Well, as you can guess, I run inside an Android app's process when I'm used. But they need some way to interact with me. So the Android engineers built be a nice Object wrapper called **SQLiteDatabase**. Once you get an instance of me and my wrapper, you've got a fully functional SQLiteDatabase at your disposal. Literally, I'm all yours!

Head First: That's just fantastic. The power of a rock solid, fully featured, yet small footprint database built into every Android app. It's a beautiful thing.

SQLiteDatabase: Can't argue with you there.

Head First: Well, thanks for joining us SQLiteDatabase. That's all the time we have, but I'm *sure* I'll be seeing you around!



Database Helper Magnets

Below is the empty implementation of TimeListDatabaseHelper and it's internal SQLiteOpenHelper implemenation TimeTrackerOpenHelper. Using the magnets below, complete the implementation using constants and string concatenation for all helper methods. .

```
public class TimeListDatabaseHelper {
```

Put constants here for table names, database version, etc.

```
        public TimeListDatabaseHelper(Context context) {

        }
```

Call super here passing in constants instead of raw values.

```
database = openHelper.getWritableDatabase();
private static final int DATABASE_VERSION = 2;
private static final String DATABASE_NAME = "timetracker.db";
+ TIMETRACKER_COLUMN_ID + " INTEGER PRIMARY KEY, "
private SQLiteDatabase database;
public static final String TIMETRACKER_COLUMN_TIME = "time";
public static final String TIMETRACKER_COLUMN_NOTES = "notes";
openHelper = new TimeTrackerOpenHelper(context);
onCreate(database);
```

```
private class TimeTrackerOpenHelper extends SQLiteOpenHelper {

    TimeTrackerOpenHelper(Context context) {
                                        // Call super on the open helper, passing in constants.
    }

    public void onCreate(SQLiteDatabase database) {
                                        // Create the database here, also using constants for the execSQL call.
    }

    public void onUpgrade(SQLiteDatabase database,
            int oldVersion, int newVersion) {
                                        // Drop and recreate the database tables down here...
    }
}
}
```

```
public static final String TIMETRACKER_COLUMN_ID = "id";
private static final String TABLE_NAME = "timerecords";
super(context, DATABASE_NAME, null, DATABASE_VERSION);
+ TIMETRACKER_COLUMN_TIME + " TEXT, "
"CREATE TABLE " + TABLE_NAME + "( "
);
database.execSQL(
database.execSQL("DROP TABLE IF EXISTS " + TABLE_NAME);
private TimeTrackerOpenHelper openHelper;
+ TIMETRACKER_COLUMN_NOTES + " TEXT )"
```



Database Helper Magnets Solution

Below is the implementation of TimeListDatabaseHelper and it's internal SQLiteOpenHelper implemenation TimeTrackerOpenHelper. You should have completed the implementation using constants and string concatenation for all helper methods.

```
public class TimeListDatabaseHelper {

    private static final int DATABASE_VERSION = 2;
    private static final String DATABASE_NAME = "timetracker.db";
    private static final String TABLE_NAME = "timerecords";

    public static final String TIMETRACKER_COLUMN_ID = "id";
    public static final String TIMETRACKER_COLUMN_TIME = "time";
    public static final String TIMETRACKER_COLUMN_NOTES = "notes";
```

All of the constants for referencing the database internal values.

```
    private TimeTrackerOpenHelper openHelper;
    private SQLiteDatabase database;
```

Store variables for the OpenHelper and the database it opens

```
    public TimeListDatabaseHelper(Context context) {
        openHelper = new TimeTrackerOpenHelper(context);
        database = openHelper.getWritableDatabase();
    }
```

Get the writable database from the open helper.

```
private class TimeTrackerOpenHelper extends SQLiteOpenHelper {

    TimeTrackerOpenHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }

    public void onCreate(SQLiteDatabase database) {
        database.execSQL(
            "CREATE TABLE " + TABLE_NAME + "( "
                + TIMETRACKER_COLUMN_ID + " INTEGER PRIMARY KEY, "
                + TIMETRACKER_COLUMN_TIME + " TEXT, "
                + TIMETRACKER_COLUMN_NOTES + " TEXT )"
        );
    }

    public void onUpgrade(SQLiteDatabase database,
            int oldVersion, int newVersion) {
        database.execSQL("DROP TABLE IF EXISTS " + TABLE_NAME);
        onCreate(database);
    }
  }
}
```

Build the database table.

Drop and recreate the table on upgrade.

You can implement save with execSQL...

Now that you have a clean encapsulated helper class for managing your database, let's implement saving time records into the database. Start by adding a method to `TimeListDatabaseHelper` to save a time record called `saveTimeRecord`.

Passing in the time and notes values as input parameters and constructing a SQL statement using string concatenation, you could write this method.

```
public void saveTimeRecord(String time, String notes) {
        database.execSQL("INSERT INTO TIMERECORDS"
            + " (TIME, NOTES)"
            + " VALUES ('" + time + "', '" + notes + "')"
        );
    }
```

The time and notes values are passed into the save method as input parameters.

The input parameters are properly escaped and concatenated in the SQL statement.

Note the spaces at the beginnings of the lines. Without proper spacing the SQL statement will throw an error.



TimeTracker DatabaseHelper.java



Be careful with execSQL and raw SQL strings.

SQL statements in your code are not checked by the compiler. So if you have errors in your SQL statements, you won't know until you run them. In many ways, these dynamic SQL stataments where you're concatenating multiple strings at runtime are even worse! At least with complete SQL statement strings you can visually inspect the SQL statements for accuracy. Dynamically generating SQL statements at runtime can be quite difficult to debug.

... but it's a lot better to use insert

Knowing that dynamically creating SQL statements to execute at runtime can be quite difficult, Android provides a number of utilities to help you avoid this.

One of these utilities is the `insert` method on `SQLiteDatabase`. `Insert` takes a parameter called `ContentValues` consisting of a set of key/value pairs consisting of the table column name and the value to insert.

```
public void saveTimeRecord(String time, String notes) {
    ContentValues contentValues = new ContentValues();

    contentValues.put(TIMETRACKER_COLUMN_TIME, time);
    contentValues.put(TIMETRACKER_COLUMN_NOTES, notes);

    database.insert(TABLE_NAME, null, contentValues);
}
```



TimeTracker
DatabaseHelper.java

there are no Dumb Questions

Q: Does executing an insert from a raw SQL function work?

A: Yes, it works just fine. You can execute arbitrary SQL statements using execSQL.

Q: OK, so I could use either. What makes the insert method so much better?

A: There are a few things that make the insert method much better to use. First of all, you don't have to worry about the syntax to combine the strings. With execSQL, you have to combine the insert and the database name with the fields you're inserting in to, plus the values. And all this has to be properly formatted with spaces, commas, parentheses, and other formatting.

Q: So I don't have to do any of that formatting with insert?

A: Correct. You're passing the same information, but organized in a data structure rather than a raw String. This helps you avoid a lot of the nastiness of piecing together all of these bits of Strings in SQL statements.

Add database access to TimeTracker

Now that you have a database setup and configured to save time records, you can start saving times entered in the app. Start by removing the `TimeTrackerOpenHelper` from the `TimeTracker` and replace it with an instance of `TimeTrackerOpenHelper` with a member variable to reference later.

```
public class TimeTracker extends Activity {
    private TimeTrackerAdapter timeTrackerAdapter;
    private TimeTrackerDatabaseHelper databaseHelper;
```

Create a member variable for the database helper.

```
    public static final int TIME_ENTRY_REQUEST_CODE = 1;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        ListView listView = (ListView)findViewById(R.id.times_list);
        timeTrackerAdapter = new TimeTrackerAdapter();
        listView.setAdapter(timeTrackerAdapter);
```

Remove the open helper.

~~TimeTrackerOpenHelper openHelper = new TimeTrackerOpenHelper(this)~~;

```
        databaseHelper = new TimeTrackerDatabaseHelper(this);
```

Instantiate the database helper.



TimeTracker.java

Save new times to the database

By adding the `TimeTrackerDatabaseHelper` to the `TimeTracker` Activity, you have access to the database and you can start saving times.

You're already saving times to the `TimeTrackerAdapter` in `onActivityResult`. Leave that code for now and add an additional call in `onActivityResult` to save the new time. Since the database helper is in view, just add a call to `addTimeRecord` with the new data after adding it tp the list adapter.

```
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
        if (requestCode == TIME_ENTRY_REQUEST_CODE) {
                if (resultCode == RESULT_OK) {

                        String time = data.getStringExtra(TIME_KEY);
                        String notes = data.getStringExtra(NOTES_KEY);

                        databaseHelper.saveTimeRecord(time, notes);

                        timeTrackerAdapter.addTimeRecord(new TimeRecord(time, notes));
                        timeTrackerAdapter.notifyDataSetChanged();
                }
        }
}
```

Save the newly entered time to the database by calling saveTimeRecord on your database helper.



TimeTracker.java

Now let's get rid of some dead code before testing it out....

Remove old code

Before you run your new code to save new times to the database, take a minute to clean up the old ,unused code you have in the app.

Start by deleting the `TimeTrackerOpenHelper` from your project since you've moved your `SQLiteOpenHelper` implementation to inside the `TimeTrackerDatabaseHelper`.



TimeTracker OpenHelper.java



You can also remove the code that adds the hard coded TimeRecords to the adapter. They were only needed since you didn't have data persistence. Now that you're storing times in the database, this will just be confusing.

```
public TimeTrackerAdapter() {
    times.add(new TimeRecord(
          "38:23", "Feeling good!"));
    times.add(new TimeRecord(
          "49:01", "Tired. Needed more caffeine."));
    times.add(new TimeRecord(
          "26:21", "I totally rocked it!"));
    times.add(new TimeRecord(
          "29:42", "Lost some time on the hills. But pretty good."));
}
```

Remove all of the test code adding hard coded TimeRecords in the adapter.



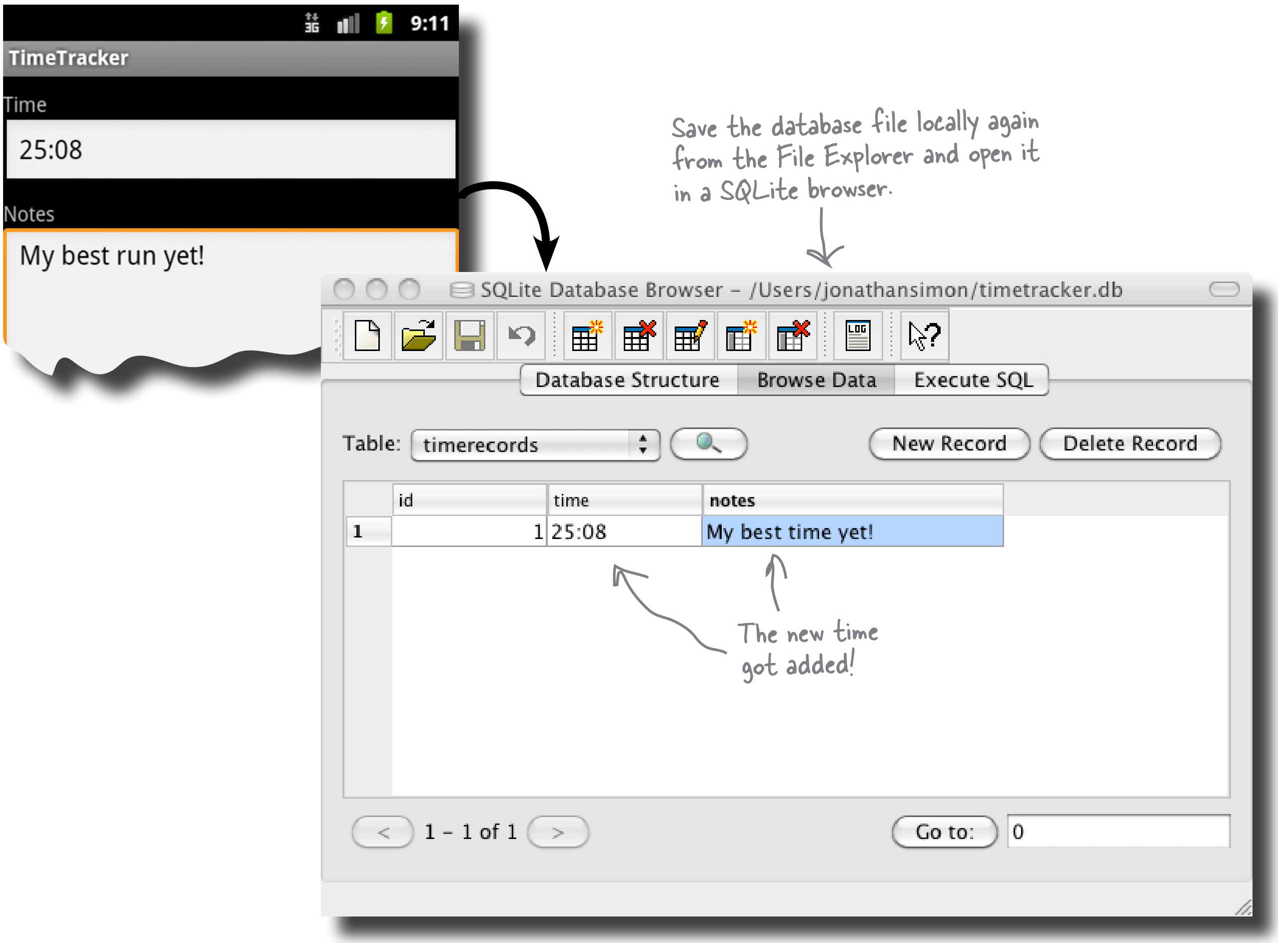
TimeTracker Adapter.java



Test Drive

Now run the app and add a new time. With your latest changes to the `TimeTracker`, you'll save to the new time to the database as well as the `TimeTrackerAdapter`.

You won't see the database changes directly in the app. You'll be able to do this later, once you connect the ListView to display results directly from the database. Meanwhile, you can view the data in the database directly and see that the new record is there.



Query the database

It's great that the time record is saving to the database, but in order to use the stored information, you need to be able to query the database. Just like `execSQL`, `SQLiteDatabase` has a method called `rawQuery` that allows you to execute raw String based SQL queries on the database.

Now add a method called getAllTimeRecords to `TimeTrackerDatabaseHelper` that will query the database for all time records. This method will execute a select all query against the database to return all of the rows in the `timerecords` table.

```
public TimeTrackerDatabaseHelper(Context context) {
        openHelper = new TimeTrackerOpenHelper(context);
        database = openHelper.getWritableDatabase();
}

public void saveTimeRecord(String time, String notes) {
        ContentValues contentValues = new ContentValues();
        contentValues.put(TIMETRACKER_COLUMN_TIME, time);
        contentValues.put(TIMETRACKER_COLUMN_NOTES, notes);
        database.insert(TABLE_NAME, null, contentValues);
}

public Cursor getAllTimeRecords() {
        return database.rawQuery(
                "select * from " + TABLE_NAME,
                null
        );
}
```

This selects all of the rows.

There are no selection args since you're selecting all of the records.



TimeTracker DatabaseHelper.java

SQLite queries return cursors

A `Cursor` is an object wrapper around a database result set. The `Cursor` contains columns and rows filled with data. Think of it as a mini spreadsheet with utility methods to navigate the results and retrieve specific data values.

The columns from the database

The database query returns a Cursor which is being passed to the caller of getAllTimeRecords.

id	time	notes
1	38:23	Feeling good!
2	49:01	Tired. Needed more caffeine!
3	26:21	I really rocked it!
4	29:42	Lost some time on this hills. But pretty good.

The rows are the data returned from the query. Your query is returning all of the data, but a more specific query may only return a smaller set.



Geek Bits

There are some disadvantages of using rawQuery just like using the raw executeSQL method. For a simple select all method, this works, but for more complicated queries where you'll be concatenating string values for column names and specific search criteria, this approach falls short. But just like the insert method, SQLiteDatabase has a several query helper methods to simplify complex database queries.

Navigating the cursor...

Now you've queried the database and gotten a `Cursor` returned. Now let's take a look at how to navigate the `Cursor` and retrieve data values.

When you work with a spreadsheet, you have a selected row and column which brings a cell into focus. The `Cursor` works the same way.

The `Cursor` keeps track of a selected row internally and includes several methods to update the `Cursor`'s selected row.



The selected row.

id	time	notes
1	38:23	Feeling good!
2	49:01	Tired. Needed more caffeine!
3	26:21	I really rocked it!
4	29:42	Lost some time on this hills. But pretty good.

Think of this whole row in focus.

Row position management methods

`moveToFirst()`

`moveToNext()`

`moveToPosition(int position)`

Row selection metadata methods

`isFirst()`

`isLast()`

`getCount()`

`getPosition()`



Make sure to set the cursor row before retrieving values.

Cursors start out with the selected row set to -1. So if you try and retrieve a value based on that row, you'll get a nasty exception. Make sure to call `moveToFirst` or `moveToPosition` before attempting to retrieve a value.

... and retrieving values

Once the desired row is selected, you can retrieve data values using separate getter methods for each data type.

Data retrieval methods

```
getFloat(int columnIndex)
getInt(int columnIndex)
getBlob(int columnIndex)
getString(int columnIndex)
getDouble(int columnIndex)
```

Looking at this sample data set, if you move the cursor to the first row and then call **getString(1)**, it will return the String **"38:23"**.

Calling getString(1) retreives the value as a String from the column at index 1.

The selected row.



id	time	notes
1	38:23	Feeling good!
2	49:01	Tired. Needed more caffeine!
3	26:21	I really rocked it!
4	29:42	Lost some time on this hills. But pretty good.

there are no Dumb Questions

Q: How do I know which type getter to use?

A: When you create your database, you assign a column type to each column. You can use whichever type you assigned to the column for the getter.

Q: What happens if I pick the wrong type?

A: Android does it's best to convert what's stored in the database to the type of the getter method you called. If it cant do the conversion it will throw an exception.

Iterating the cursor

Sometimes you just need to get a single value from the cursor. In those cases, you can go straight to the row and get the value you need. Very often though, you'll be *iterating* through a number of results and processing them in bulk.

```
Cursor cursor = helper.getTimeRecordList();

if (cursor.moveToFirst()) {
      do {
            String time = cursor.getString(1);
            String notes = cursor.getString(2);
            Log.d("DB Value", time + " " + notes);
      } while (cursor.moveToNext());
}

if (!cursor.isClosed()) {
      cursor.close();
}
```

Query the database using the helper. (`helper.getTimeRecordList()`)

Move the cursor to the first row, checking the boolean response before continuing. (`cursor.moveToFirst()`)

Retrieve the data values from time and notes columns and print the value. (the three lines inside `do`)

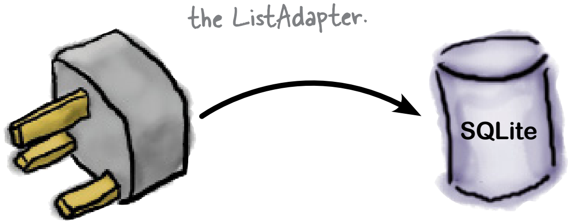
Move to next if there are more rows. (`cursor.moveToNext()`)

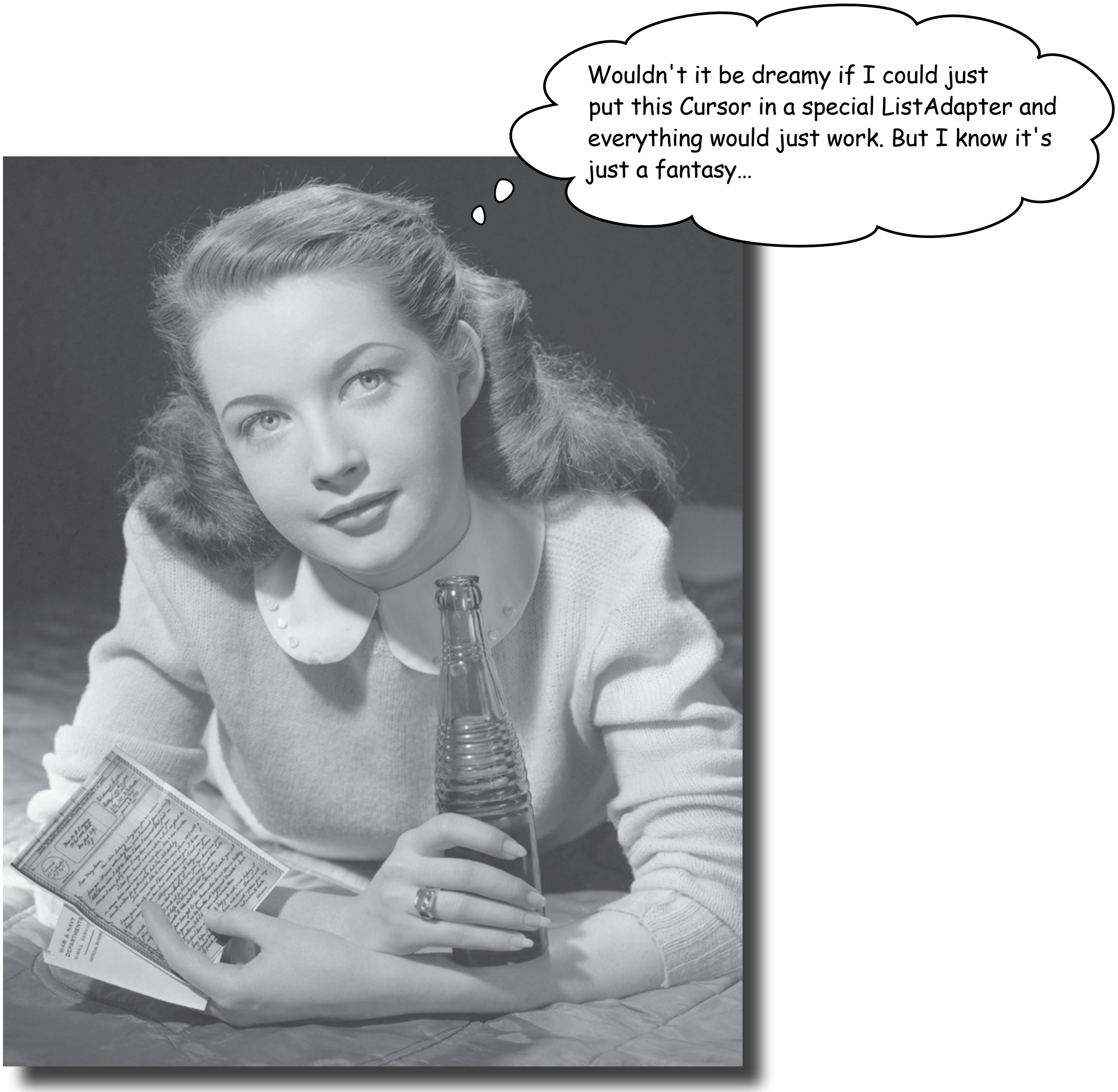
Always make sure to close the cursor when you're done. (`cursor.close()`)

Next steps

Now you have data saving in the database, a query to retrieve the `Cursor`, and a way to iterate the `Cursor` to get specific values. Now you need to get the data from the Cursor into your `ListAdapter`

Up next: Connecting the SQLite database and the ListAdapter.

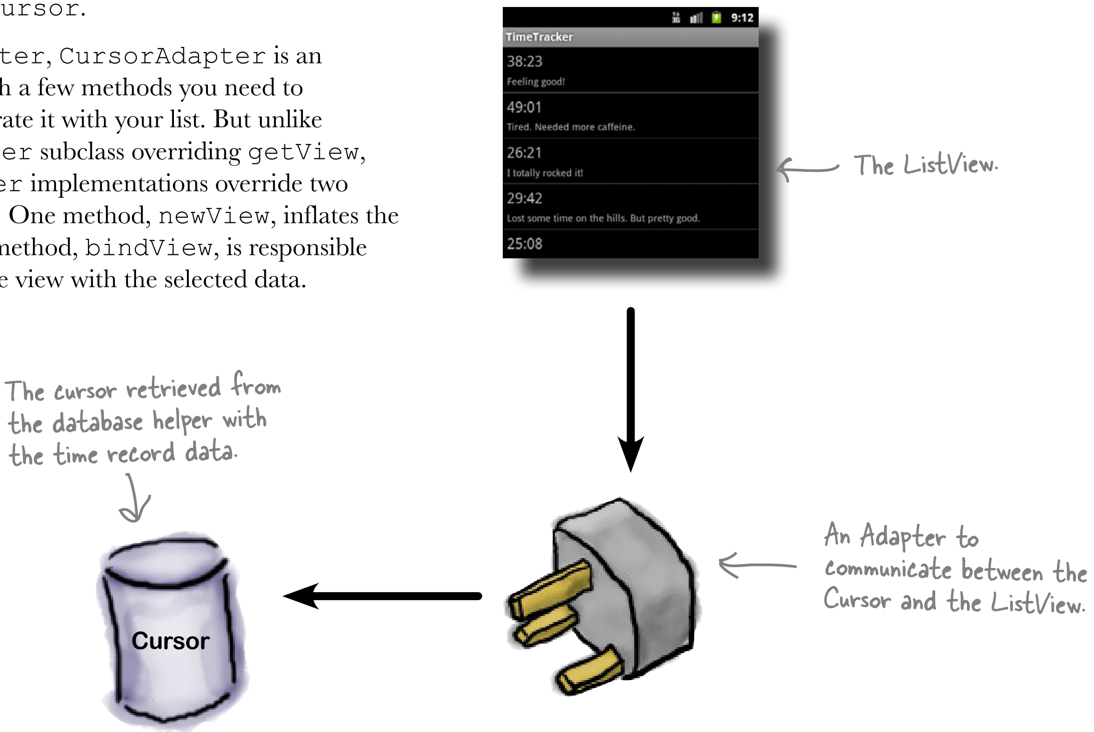



Wouldn't it be dreamy if I could just put this Cursor in a special ListAdapter and everything would just work. But I know it's just a fantasy...

Use CursorAdapter

The Android SDK includes a special adapter to easily get a `Cursor` working with a `ListView` called `CursorAdapter`. You'll be able to instantiate a `CursorAdapter`, passing in a `Cursor`. The `CursorAdapter` then acts as a facilitator between the `ListView` and the `Cursor` to render the contents of the `Cursor`.

Like `BaseAdapter`, `CursorAdapter` is an Abstract class with a few methods you need to override to integrate it with your list. But unlike the `BaseAdapter` subclass overriding `getView`, `CursorAdapter` implementations override two separate method. One method, `newView`, inflates the view. The other method, `bindView`, is responsible for populating the view with the selected data.



there are no Dumb Questions

Q: Do I have to use CursorAdapter?

A: You could follow the idea from a few pages back and implement the CursorAdapter on your own. Unless you have a ***really*** good reason though, you should just use CursorAdapter. It will save you a lot of headaches getting going

Q: It looks like the getView implementation is split out into these two methods newView and bindView. Do I have to impelement getView as well?

A: No. In fact you shouldn't. Just implement newView and getView and you'll be all set!



Cursor Adapter Magnets

Below is the updated `TimeTrackerAdapter` extending `CursorAdapter`. Implement `newView` to create the view and `bindView` to populate the view with data. The `cursor` manages all iteration, so you just need to call the getter value methods and render the results.

```
public class TimeTrackerAdapter extends CursorAdapter {

    public TimeTrackerAdapter (Context context, Cursor cursor) {
        super(context, cursor);
    }

    public void bindView(View view, Context context, Cursor cursor) {




    }

    public View newView(Context context, Cursor cursor, ViewGroup parent) {




    }
}
```

The adapter now extends CursorAdapter.

Pass the cursor to super.

Add a Cursor param to the constructor.

The adapter handles all cursor iteration for you, you just need to display the values in the selected row.

You can use the same view for the display. Just create an inflater and inflate the view.

```
valueTextView.setText(cursor.getString(cursor.getColumnIndex(2));
```

```
LayoutInflater inflater = LayoutInflater.from(parent.getContext());
```

```
TextView valueTextView = (TextView) view.findViewById(R.id.notes_view);
```

```
nameTextView.setText(cursor.getString(cursor.getColumnIndex(1));
```

```
View view = inflater.inflate(R.layout.list_item, parent, false);
```

```
return view;
```

```
TextView nameTextView = (TextView) view.findViewById(R.id.time_view);
```



Cursor Adapter Magnets Solution

Below is the updated `TimeTrackerAdapter` extending `CursorAdapter`. You should have implemented `newView` to create the view and `bindView` to populate the view with data. The `cursor` manages all iteration, so you just needed to call the getter value methods to render the results.

```
public class TimeTrackerAdapter extends CursorAdapter {

    public TimeTrackerAdapter (Context context, Cursor cursor) {
        super(context, cursor);
    }

    public void bindView(View view, Context context, Cursor cursor) {
        TextView nameTextView = (TextView) view.findViewById(R.id.time_view);
        nameTextView.setText(cursor.getString(cursor.getColumnIndex(1));

        TextView valueTextView = (TextView) view.findViewById(R.id.notes_view);
        valueTextView.setText(cursor.getString(cursor.getColumnIndex(2));
    }

    public View newView(Context context, Cursor cursor, ViewGroup parent) {
        LayoutInflater inflater = LayoutInflater.from(parent.getContext());
        View view = inflater.inflate(R.layout.list_item, parent, false);
        return view;
    }
}
```

The time and notes fields are both retrieved and populated with data from getString calls to the cursor.

The LayoutInflater is retrieved and the layout in inflated and returned.

Update TimeTracker

The `TimeTrackerAdapter` is now updated to be a `CursorAdapter` subclass. The last thing you need to do to use it is to update the `TimeTracker` Activity to use it. Start by passing in the context (`this`) and the `Cursor` containing the list of time records to the new adapter.

```
public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        databaseHelper = new TimeTrackerDatabaseHelper(this);

        ListView listView = (ListView)findViewById(R.id.times_list);
        timeTrackerAdapter = new TimeTrackerAdapter(
                this, databaseHelper.getTimeRecordList());
        listView.setAdapter(timeTrackerAdapter);
}
```

Pass in the Cursor and the context to the adapter.

Now remove the call to add a time record to the adapter. You're already adding the time record to the database.

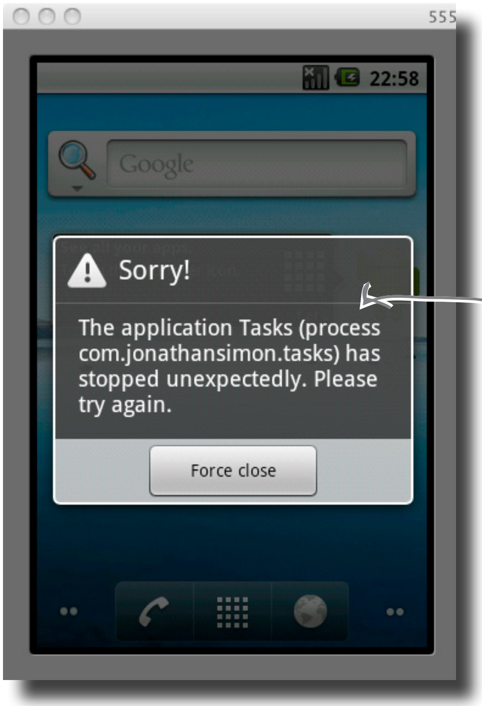
```
protected void onActivityResult(int requestCode,
        int resultCode, Intent data) {
        if (requestCode == TIME_ENTRY_REQUEST_CODE) {
                if (resultCode == RESULT_OK) {
                        String time = data.getStringExtra(TIME_KEY);
                        String notes = data.getStringExtra(NOTES_KEY);
                        databaseHelper.saveTimeRecord(time, notes);
                        timeTrackerAdapter.changeCursor(
                                databaseHelper.getTimeRecordList());
                        timeTrackerAdapter.addTimeRecord(time, notes);
                        timeTrackerAdapter.notifyDataSetChanged();
```

Save the new time record in the database, and update the cursor in the adapter.

Don't add the time record to the adapter anymore or call the data change notification. (The last two lines above are struck through.)

Test Drive

The `TimeTrackerAdapter` is now updated to a `CursorAdapter` and connected to the `ListView` from the `TimeTracker` Activity. Go ahead and run the app. There is one time record stored in the database, so if everything works, you should see it in the list.



Something's going wrong in your code.

Uh oh! Looks like there's an error in the code.



Tracking down the error

Open LogCat. If you closed it, you can reopen it by going to Window → Show View → Other, opening the Android folder and selecting LogCat.

The exception stack trace printing out in LogCat

```
Caused by: java.lang.IllegalArgumentException: column '_id' does not exist
    at android.database.AbstractCursor.getColumnIndexOrThrow(AbstractCursor.java:314)
    at android.widget.CursorAdapter.init(CursorAdapter.java:111)
    at android.widget.CursorAdapter.<init>(CursorAdapter.java:90)
    at com.headfirstlabs.timetracker.TimeTrackerAdapter.<init>(TimeTrackerAdapter.java:14)
    at com.headfirstlabs.timetracker.TimeTracker.onCreate(TimeTracker.java:27)
    at android.app.Instrumentation.callActivityOnCreate(Instrumentation.java:1047)
    at android.app.ActivityThread.performLaunchActivity(ActivityThread.java:1586)
```

If you look at the error, you'll see the following error message "`Caused by: java.lang.IllegalArgumentException: column '_id' does not exist`". At first glace, it might seem strange as you have an `id` column in your database.

But look a little closer, and you'll see it's not looking for a column called `id`, it's actually looking for a column called `_id` with an underscore in front.

The class overview in the online docs for CursorAdapter even specifies that you need an _ID column.

Class Overview

Adapter that exposes data from a [Cursor] to a [ListView] widget. The Cursor must include a column named "_id" or this class will not work.

BRAIN POWER

Now that you know the problem, how are you going to fix it? Think about *all of the steps* you would take to implement the fix before going on.

Long Exercise

Below is the current complete code for the `TimeTrackerDatabaseHelper`. All of the changes you need to make to the database to update the table to use the `_id` column (with the underscore) instead of the `id` column (without an underscore) is in this class.

```
package com.headfirstlabs.timetracker;

import android.content.ContentValues;
import android.content.Context;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;

public class TimeTrackerDatabaseHelper {
      private static final int   DATABASE_VERSION = 2;
      private static final String DATABASE_NAME = "timetracker.db";
      private static final String TABLE_NAME = "timerecords";

      public static final String TIMETRACKER_COLUMN_ID = "id";
      public static final String TIMETRACKER_COLUMN_TIME = "time";
      public static final String TIMETRACKER_COLUMN_NOTES = "notes";

      private TimeTrackerOpenHelper openHelper;
      private SQLiteDatabase database;

      public TimeTrackerDatabaseHelper(Context context) {
            openHelper = new TimeTrackerOpenHelper(context);
            database = openHelper.getWritableDatabase();
      }
```

```
    public void saveTimeRecord(String time, String notes) {
        ContentValues contentValues = new ContentValues();
        contentValues.put(TIMETRACKER_COLUMN_TIME, time);
        contentValues.put(TIMETRACKER_COLUMN_NOTES, notes);
        database.insert(TABLE_NAME, null, contentValues);
    }

    public Cursor getTimeRecordList() {
        return database.rawQuery("select * from " + TABLE_NAME, null);
    }

    private class TimeTrackerOpenHelper extends SQLiteOpenHelper {
        TimeTrackerOpenHelper(Context context) {
            super(context, DATABASE_NAME, null, DATABASE_VERSION);
        }

        public void onCreate(SQLiteDatabase database) {
            database.execSQL(
                "CREATE TABLE " + TABLE_NAME + "( "
                    + TIMETRACKER_COLUMN_ID + " INTEGER PRIMARY KEY, "
                    + TIMETRACKER_COLUMN_TIME + " TEXT, "
                    + TIMETRACKER_COLUMN_NOTES + " TEXT )"
            );
        }

        public void onUpgrade(SQLiteDatabase database,
            int oldVersion, int newVersion) {
            database.execSQL("DROP TABLE IF EXISTS " + TABLE_NAME);
            onCreate(database);
        }
    }
}
```



Long Exercise Solution

Below is the complete code for the `TimeTrackerDatabaseHelper`. You should have made all of the database changes needed to update the table to use the `_id` column (with the underscore) instead of the `id` column (without an underscore).

```
package com.headfirstlabs.timetracker;

import android.content.ContentValues;
import android.content.Context;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;

public class TimeTrackerDatabaseHelper {
      private static final int   DATABASE_VERSION = 2; 3;
      private static final String DATABASE_NAME = "timetracker.db";
      private static final String TABLE_NAME = "timerecords";

      public static final String TIMETRACKER_COLUMN_ID = "id"; = "_id";
      public static final String TIMETRACKER_COLUMN_TIME = "time";
      public static final String TIMETRACKER_COLUMN_NOTES = "notes";

      private TimeTrackerOpenHelper openHelper;
      private SQLiteDatabase database;

      public TimeTrackerDatabaseHelper(Context context) {
             openHelper = new TimeTrackerOpenHelper(context);
             database = openHelper.getWritableDatabase();
      }
```

Update the version number. This will cause your app to call onUpgrade, which drops and recreates the database. (The "2;" is crossed out and replaced with "3;")

Change the "id" column to "_id" (The "id" is crossed out and replaced with = "_id";)

These changes are subtle, but really important

```
    public void saveTimeRecord(String time, String notes) {
        ContentValues contentValues = new ContentValues();
        contentValues.put(TIMETRACKER_COLUMN_TIME, time);
        contentValues.put(TIMETRACKER_COLUMN_NOTES, notes);
        database.insert(TABLE_NAME, null, contentValues);
    }

    public Cursor getTimeRecordList() {
        return database.rawQuery("select * from " + TABLE_NAME, null);
    }

    private class TimeTrackerOpenHelper extends SQLiteOpenHelper {
        TimeTrackerOpenHelper(Context context) {
            super(context, DATABASE_NAME, null, DATABASE_VERSION);
        }

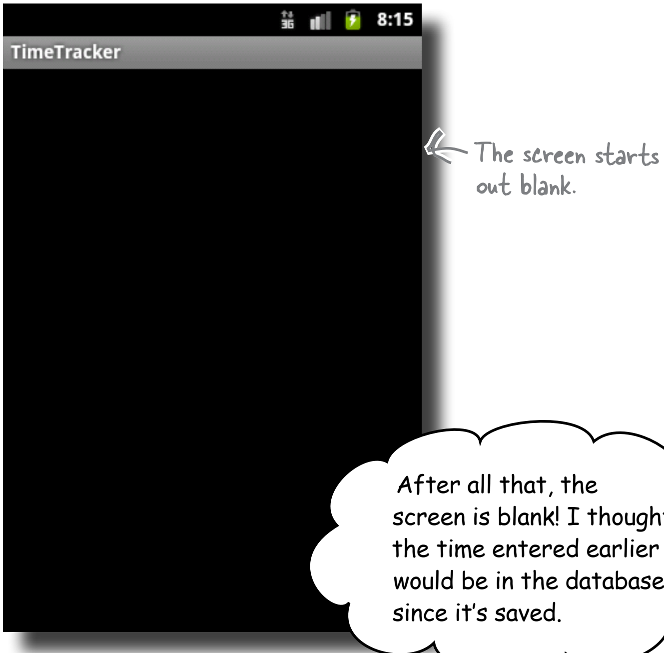
        public void onCreate(SQLiteDatabase database) {
            database.execSQL(
                "CREATE TABLE " + TABLE_NAME + "( "
                    + TIMETRACKER_COLUMN_ID + " INTEGER PRIMARY KEY, "
                    + TIMETRACKER_COLUMN_TIME + " TEXT, "
                    + TIMETRACKER_COLUMN_NOTES + " TEXT )"
            );
        }

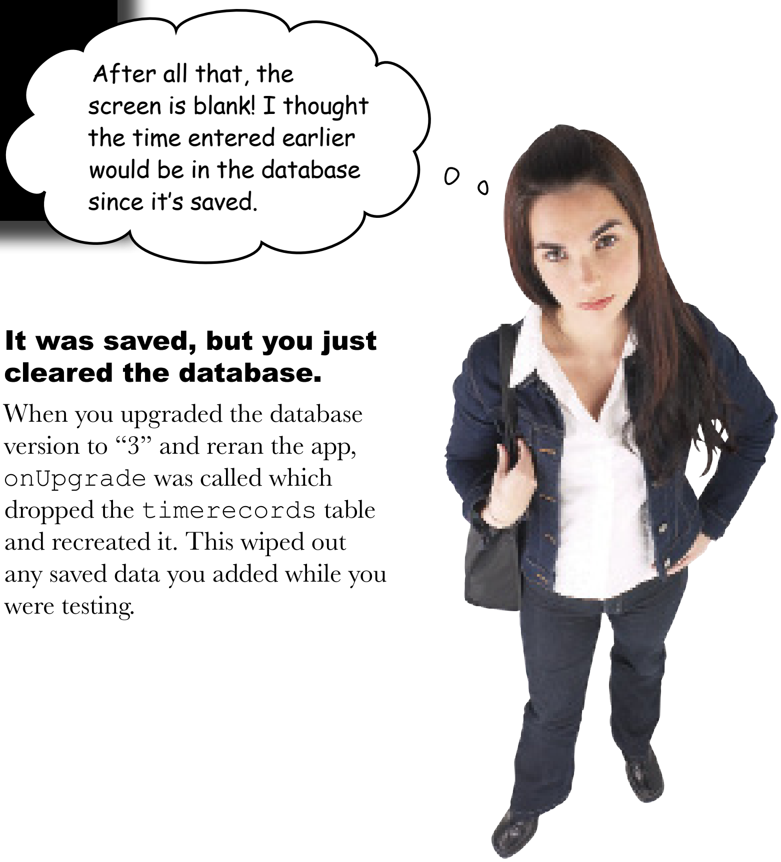
        public void onUpgrade(SQLiteDatabase database,
            int oldVersion, int newVersion) {
            database.execSQL("DROP TABLE IF EXISTS " + TABLE_NAME);
            onCreate(database);
        }
    }
}
```

You updated the database version which will drop and recreate the database, destroying all of stored data. If you found this on a production system with real users and real data, this is when you would override onUpgrade to migrate the information from the old database format to the new one.

Test Drive

Now run the app again. Since you updated the database version number, the database will be automatically wiped and recreated by the database management code when you start the app.

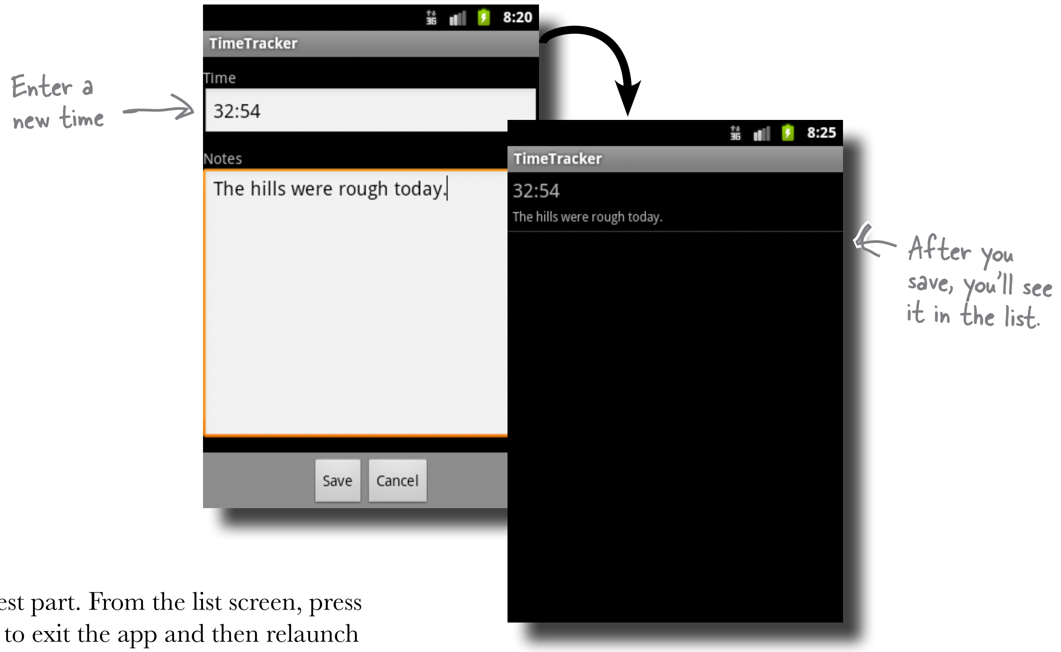




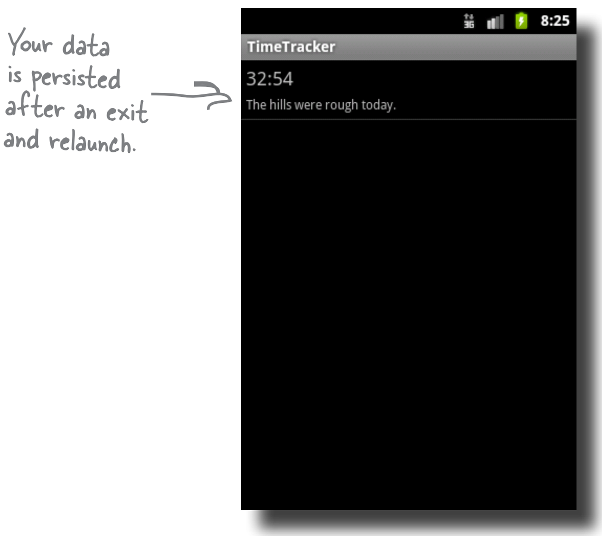
It was saved, but you just cleared the database.

When you upgraded the database version to "3" and reran the app, `onUpgrade` was called which dropped the `timerecords` table and recreated it. This wiped out any saved data you added while you were testing.

The database is starting off empty because it was just dropped and recreated. That shouldn't effect new database records though. Add a new time and save it and you should see it in the list.



But here's the best part. From the list screen, press the back button to exit the app and then relaunch it. Your data is still there!



Excellent work! Your app is now storing and loading data from a SQLite database.



Looks like another happy user!

Although there are more features you could implement in this app, you'll stop working on it here. Try implementing new features on your own, like editing and deleting time records to really take the app to the next level and make Donna even happier. But remember, don't add *too* many new features. She liked her apps to stay simple.

Have fun on your run, Donna!



Go Off Piste

Now that times are saving in the database, you're ready to move on. But if you're still thirsty for more, here are a few additional features and exercises you could work on to start honing your Android database skills.

Implement onUpgrade

You upgraded the database in this chapter without overriding onUpgrade to handle the schema change. Go back and modify the database again, implementing a data migration in onUpgrade.

Use query()

You queried the database using rawQuery(). But just like execSql, this is limited and error prone. Look into the database query() methods and implement a few more detailed queries against your database.

Implement delete and edit

Right now you have the ability to create the database and add to it. Try implementing methods on your database helper to edit previously entered time or delete them.



Your Android Toolbox

You just built your first app with full persistent SQLite database support. Use this same process to add database support to all your apps!

Cursor Iteration

- Query the Database and get a Cursor in return
- Move to a specific row location in the Cursor
- Retrieve typed data from a column
- Close the Cursor when you're done

Using Cursor Adapter

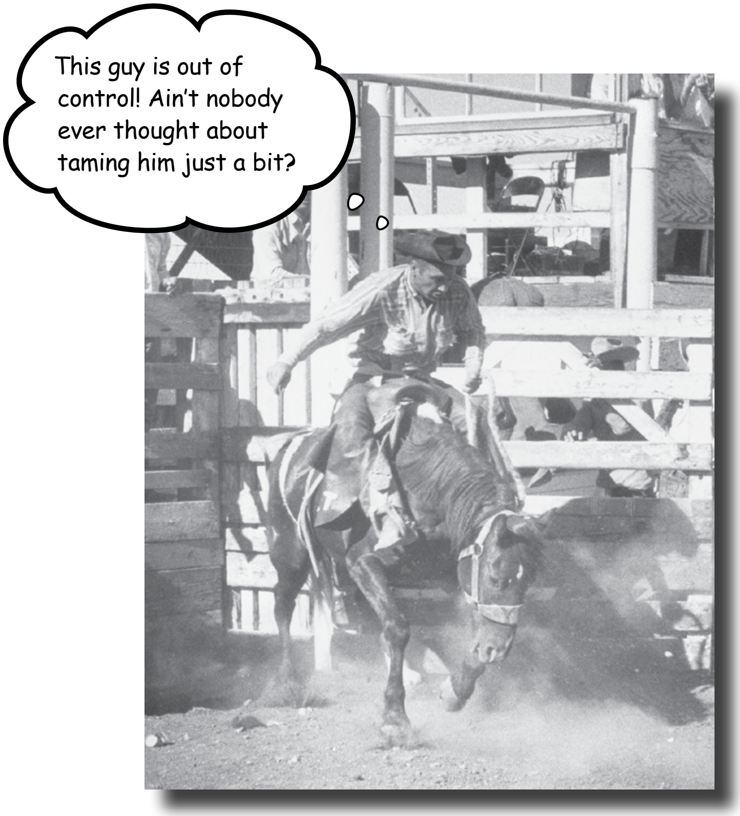
- Create a class that extends CursorAdapter
- Create a constructor that passes the Context to super, as well as a cursor
- Override newView to inflate an XML View (or create one programatically)
- Override bindView and populate the View with data from the current cursor row

BULLET POINTS

- Create your own databases for your apps so you can persist your app data.
- Use `SQLiteOpenHelper` to simplify database management.
- Wrap your `SQLiteOpenHelper` in a database helper class encapsulating your database and limiting access to it.
- Expose helper methods on the databasehelper to manage database usage throughout the app.
- Abstract constants and reusable pieces of your SQL statements to make your code resilient.
- Use `Database` helper methods for inserting and querying rather than the raw SQL methods when possible.
- Always take a look at Android's built in components (like `CursorAdapter`). They can save you a **ton** of work.
- Use `CursorAdapter` to connect your cursor to a list so you don't have to write all that Cursor management code.
- Make sure and include an "_id" column in your database if you plan to use `CursorAdapter`.
- Remember to update your database version or delete the database if you make changes to your database schema.
- If you do update your database schama, consider implementing `onUpgrade` to migrate production data.

10 relative layout

It's all relative



You've created a few screens now using LinearLayouts (and even nested LinearLayouts). But that will only get you so far. Some of the screens you'll need to build in your own apps will need to do things that you just cant' do with LinearLayout. But don't worry! Android comes with other layouts that you can use. IN this chapter, you'll learn about another super powerful layout called RelativeLayout. This allows you to layout Views on screen relative to each other (hence the name). It's new way to layout your Views, and as you'll see in the chapter, a way to optimize your screen layouts.

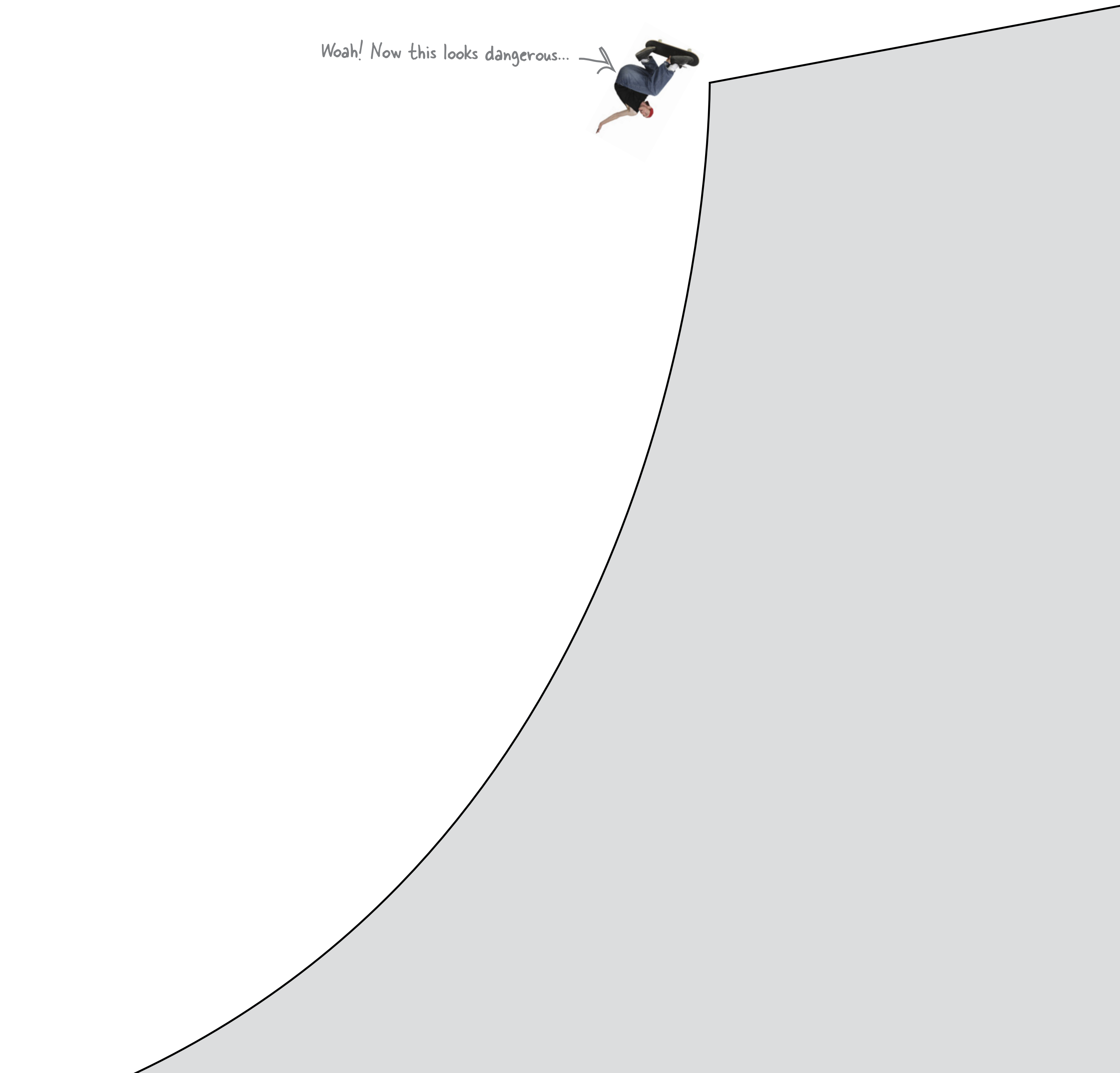
Meet Taylor and Scott, two super tight skateboarding pals

(And also dating. Well, this week anyway.)



They worry about each other when they skate apart

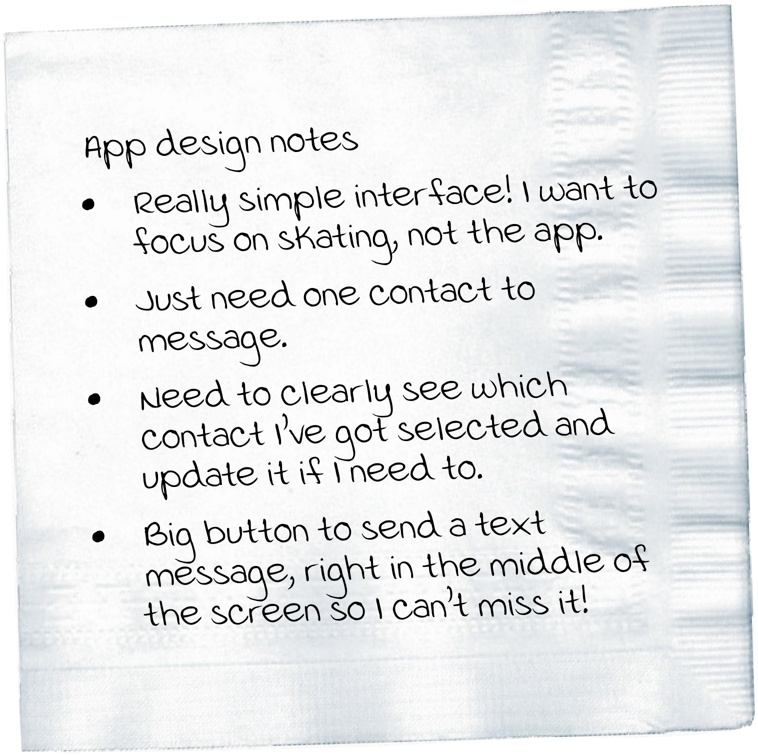
Skating can be dangerous. Crazy tricks, broken boards, cops... all kinds of things can happen! After chatting with Sam and Scott a bit, they asked you to build an app they could use to let each other know they are OK when they are skating separately.

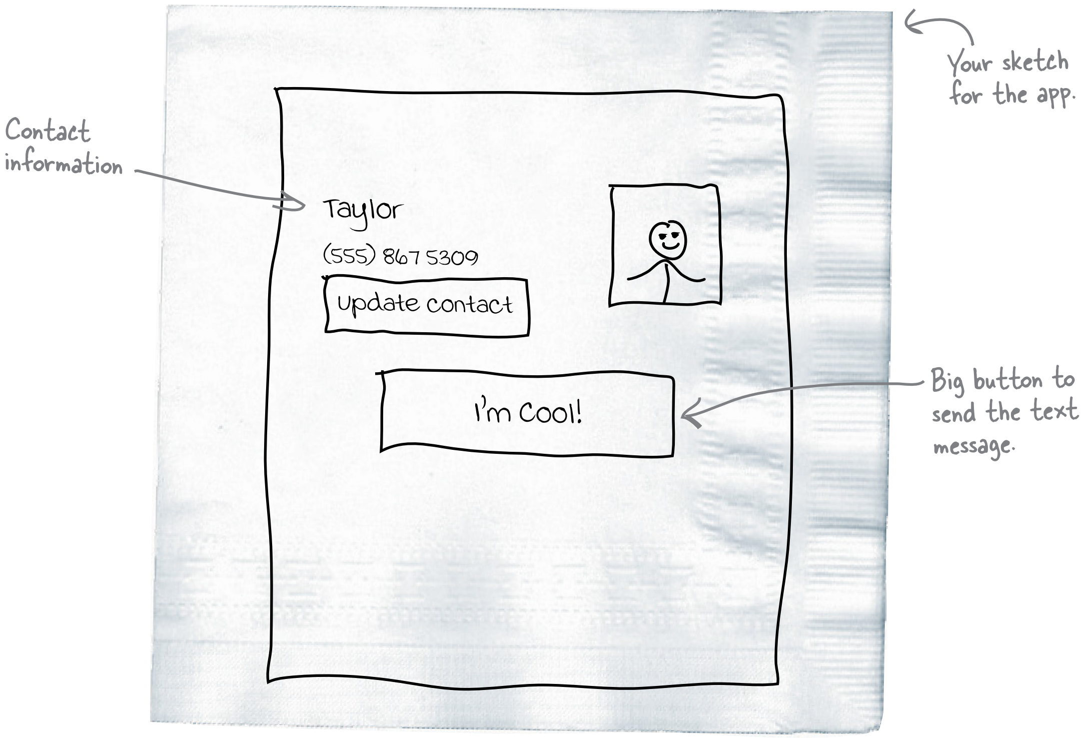

Woah! Now this looks dangerous...

Design the app

Like all good apps, building this app starts out with a good solid design. After chatting with Sam and Scott, you found out that they want a really specific app. Here are the notes from meeting with them.







Focus on the layout first

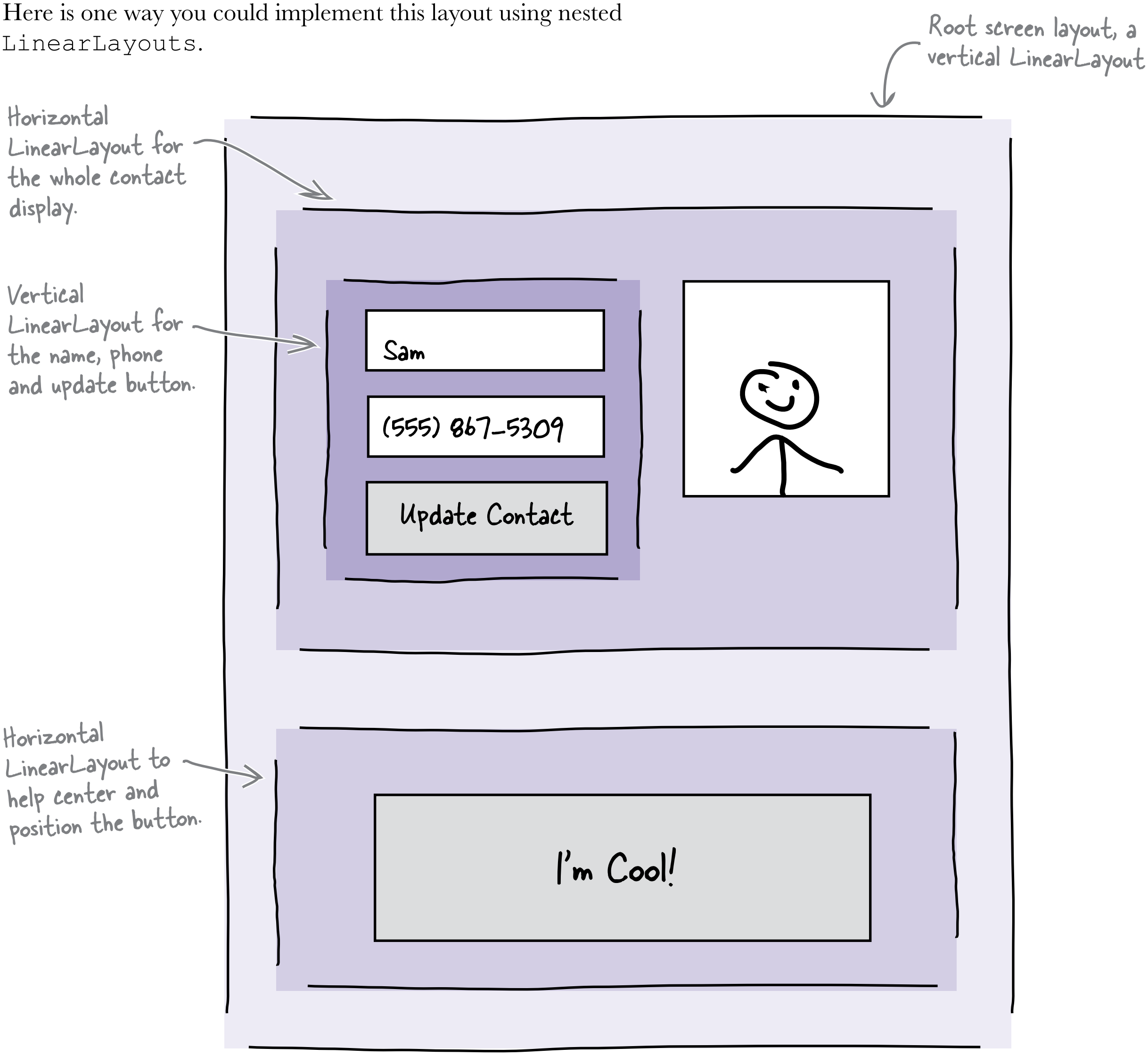
In this chapter, you'll focus on the layout. You'll learn about a new layout called `RelativeLayout` that is much more powerful then plain old `LinearLayout`.

Turn the page to get started.

Nested LinearLayout implementation

Based on the sketch, you could implement this layout using a combination of nested `LinearLayouts` (layouts inside other layouts are called nested layouts). But there's going to be a lot of nesting! And you'll need to be really careful to get all of the parameters right, like which `LinearLayouts` are veritcal, which are horizontal, how to size components and all the good stuff you've been doing with`LinearLayouts`... just a lot more at once.

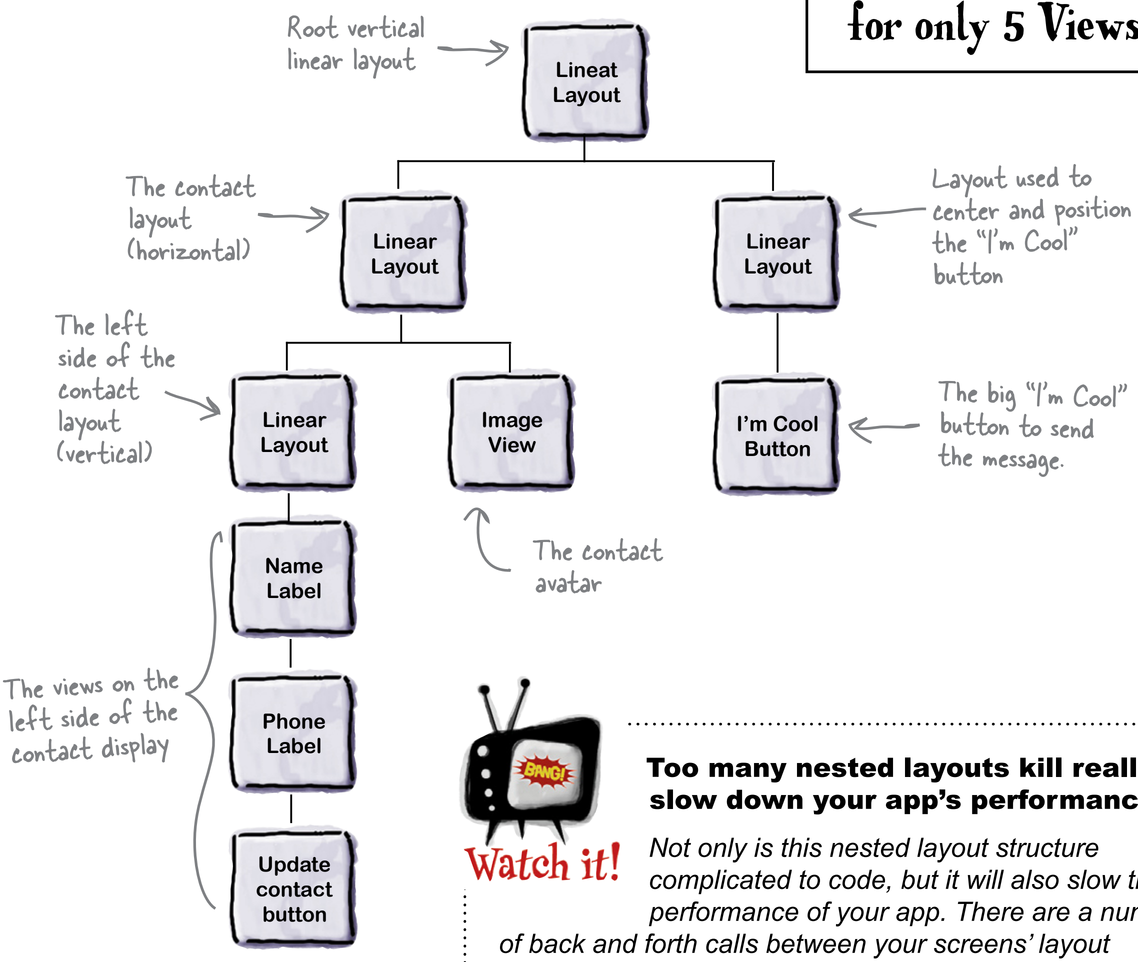
Here is one way you could implement this layout using nested `LinearLayouts`.



This is getting complicated

That's a lot of layouts! Before you start writing the code for this layout, let's take a look at the view hierarchy with the layouts and their children on a tree.

There are 4 layouts for only 5 Views!



Watch it!

Too many nested layouts kill really slow down your app's performance.

*Not only is this nested layout structure complicated to code, but it will also slow the performance of your app. There are a number of back and forth calls between your screens' layout managers and the Android layout management code, and each of these calls take time. The more layouts you have, the longer it takes to render your screens. For really complex screens, this can make a **HUGE** difference!*

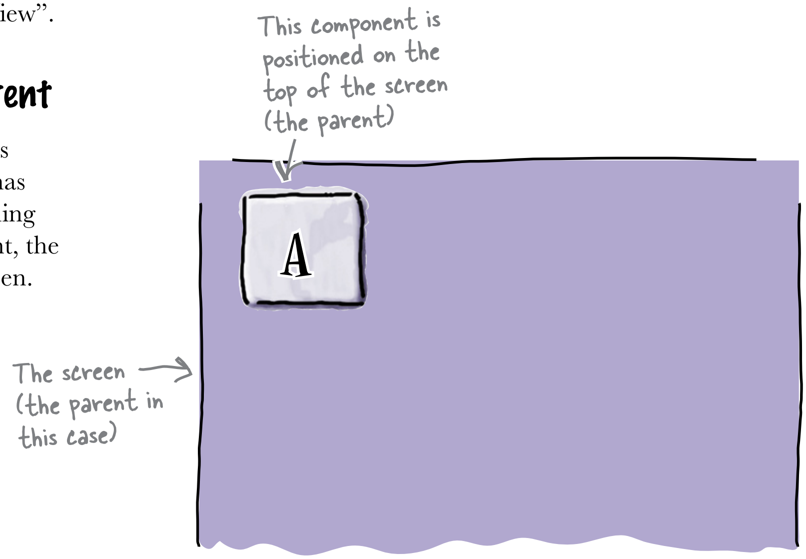
There HAS to be a better way...

Meet relative layout

`RelativeLayout` is a layout that allows you to position Views on the screen ***relative*** to each other. Where `LinearLayout` positions all Views in a line - either vertically or horizontally -`RelativeLayout` let's you express layout positions like "put this View below this other View" or "put this view to the left another View".

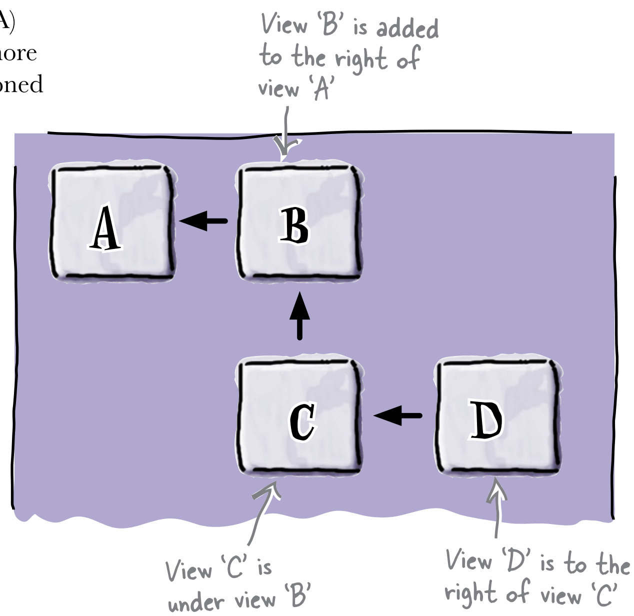
Add a view positioned in the parent

Making your own `RelativeLayout` starts with an ***anchored*** view. This is a view that has an ***anchor*** on the screen referencing something about the parent view like the top left or right, the bottom left or right, or the center of the screen.



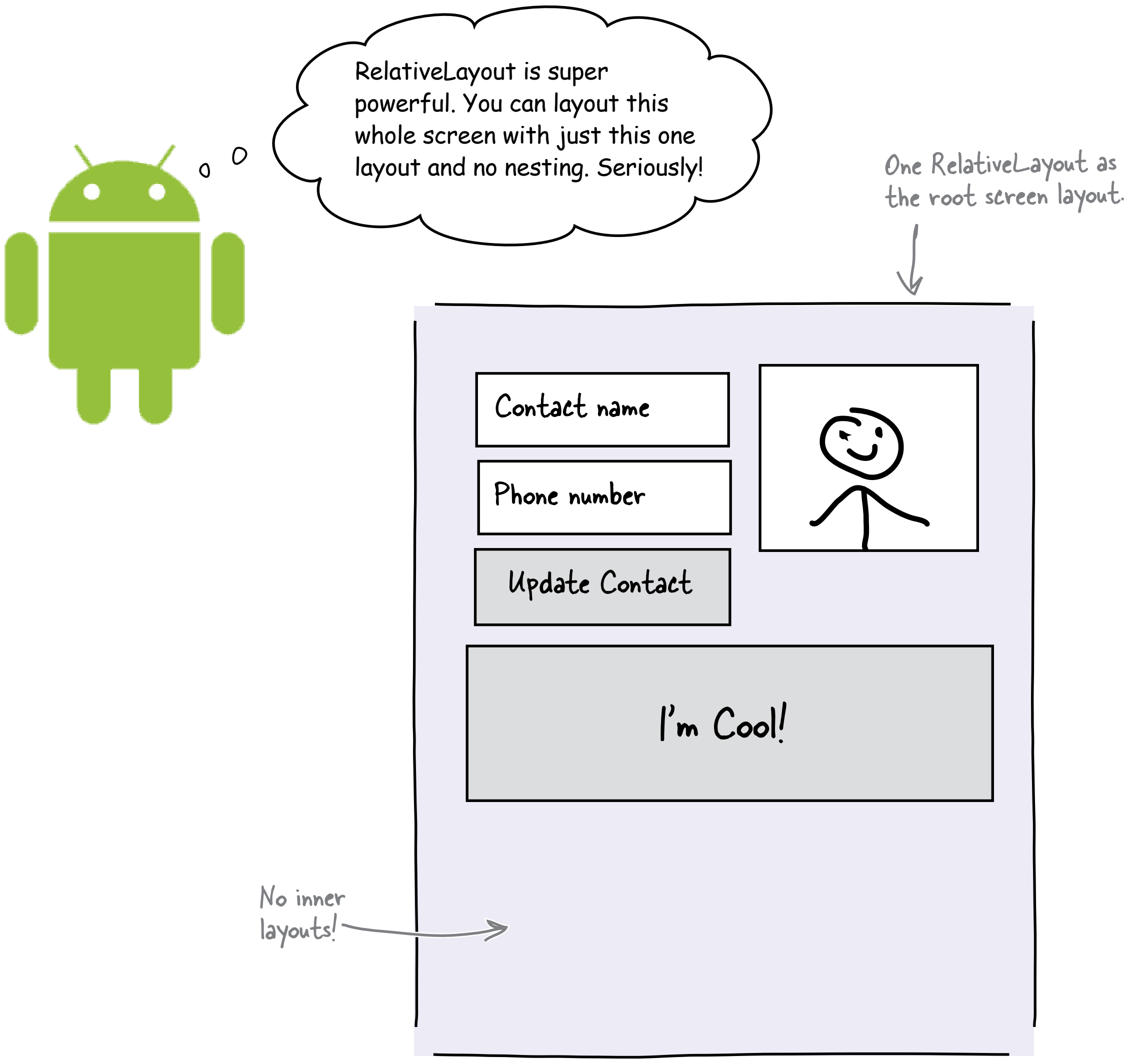
Add (a bunch) of other views

You can add (and keep adding) views positioned relative to any other view on the screen. This positioning may be relative to an anchored view (like View B positioned relative to the View A) but it doesn't have to be. You can also add more anchored views, and then other views positioned relative to that new anchor view too.



Are you ready for a challenge?

The Android layout manager thinks you can layout the entire screen using just one `RelativeLayout`. Do you believe it?

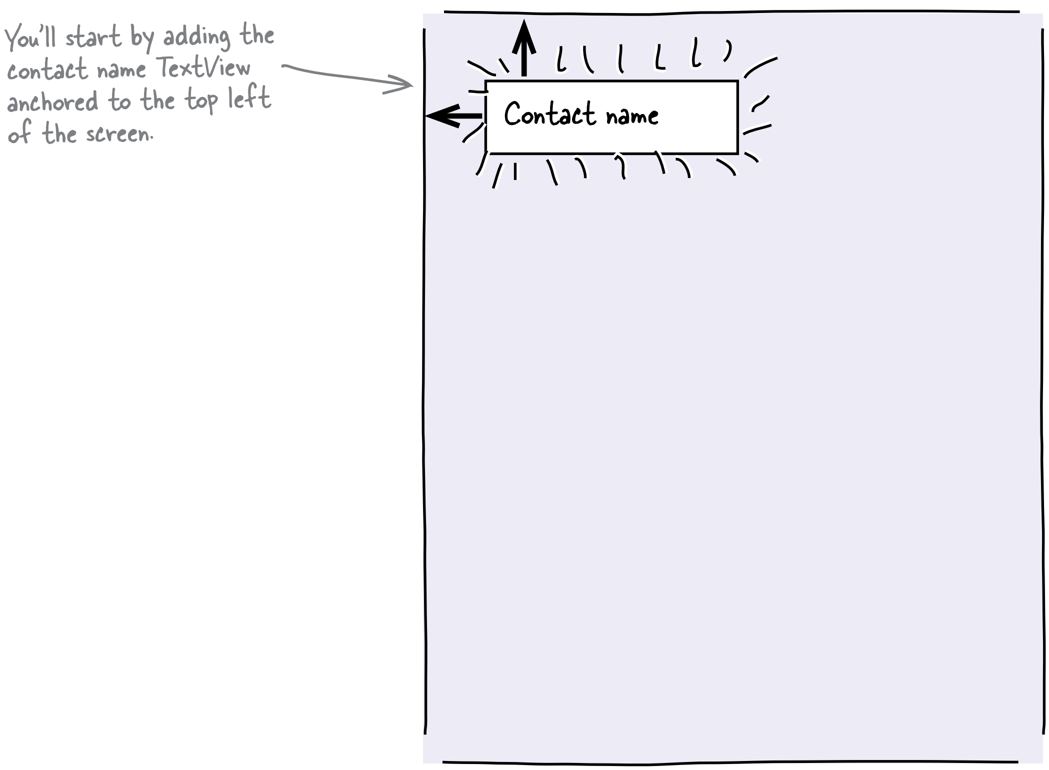


Sound impossible? Turn the page to get started laying out the screen with RelativeLayout and see for yourself!

Choose your anchor point

The first step when you make a new `RelativeLayout` is to position a View in the parent. This is a view that has an ***anchor*** on the screen referencing something about the parent view like the top, bottom, left, right or center of the screen. From there you'll position the rest of the Views relative to the first anchored view.

For the layout, the first View that you'll position in the parent is the contact Name `TextView`. And it's going to be positioned to the top left hand corner.



Once this first View is positioned, you'll be able to layout the rest of the views around it.



Anchored View Magnets

Below is the very beginnings of a `RelativeLayout`. The layout is declared with a type of `RelativeLayout` and its width and height are set to fill the screen. The `TextView` for the Contact Name is also added, but not positioned. You'll need to use the magnets with position parameters below to position the View. Remember, it should be positioned to the top left hand corner. *Hint- you can use multiple positioning attributes together.*

```
<RelativeLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent">

<TextView
        android:id="@+id/contact_name"
        android:text="Sam"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"



        />

</RelativeLayout>
```

The RelatoveLayout declaration.

The Contact Name TextView declaration without position attributes.

The positioning attributes need to go here.



main.xml

`android:layout_alignParentBottom="true"`

This attribute positions the view top the top of the parent.

`android:layout_alignParentTop="true"`

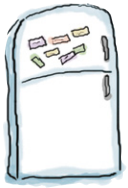
This positions the View at the top of the parent.

This positions the View to the right side of the parent.

`android:layout_alignParentRight="true"`

This positions the view to the left of the parent.

`android:layout_alignParentLeft="true"`



Anchored View Magnets Solution

Below is the very beginnings of a `RelativeLayout`. The layout is declared with a type of `RelativeLayout` and its width and height are set to fill the screen. The `TextView` for the Contact Name is also added, but not positioned. You should have used the magnets with position parameters below to position the View. It should be positioned to the top left hand corner.

```
<RelativeLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent">

<TextView
        android:id="@+id/contact_name"
        android:text="Sam"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:layout_alignParentLeft="true"
        />

</RelativeLayout>
```

This lays out the View to the TOP of the parent.

This lays out the View to the LEFT of the parent.



main.xml

These two attributes *together* position the Contact Name View at the top left of the screen.



Test Drive

Now that the first View is positioned, run the app and let's make sure the View is positioned correctly.

The View is positioned on the top left..

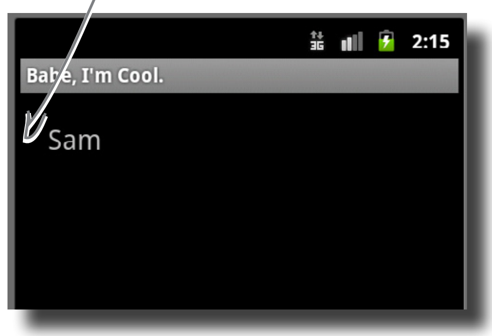


It's close, but you can make it even better. The View is in fact positioned on the top left, but it needs some space so it's not pinned to the edges. The font also needs to be a bit larger. Let's make those updates to the layout before moving on.

```
<TextView
        android:id="@+id/anchored_button"
        android:text="Sam"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:layout_alignParentLeft="true"
        android:layout_marginLeft="20dp"
        android:layout_marginTop="20dp"
        android:textSize="20dp"
        />
```

Polishing the layout with some margins and text sizing.

Now there's a little bit of space.

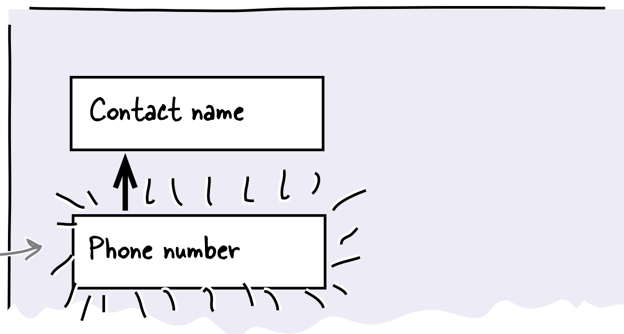


Now it's time to layout some more Views!

Positioning views relative to on screen views

The Contact Name View is looking great! Now it's time to add another View. The next view to add is the Phone Number view. You'll position it under the Contact Name view.

Next, you're going to position the phone number TextView under the Contact Name TextView.



Attributes for relative positioning to other Views

There are different layout positioning attributes for laying out Views relative to parents and relative to other Views on the screen. The Contact Name view is positioned relative to the parent, but the Phone Number View is going to be positioned relative to the Contact Name view (another view on the screen).

Position to the left of another component..

`android:layout_toLeftOf`

Position above... `android:layout_above`

`android:layout_below` Position below...

`android:layout_toRightOf`

Position to the right...

Using these attributes

You add these attributes to View declarations in the layout XML just like the other positioning attributes. The difference is that instead of using a value of true, you pass in the ID of the view you want to position your view relative to.

This attribute is added to the View you're positioning.

```
android:layout_below = "@+id/contact_name"
```

Here you supply the View you want to position this View relative to.

Add the phone number view

This snippet shows the Phone Number `TextView` in the layout positioned using the `android:layout_below` attribute to be underneath the Contact Name `TextView`.

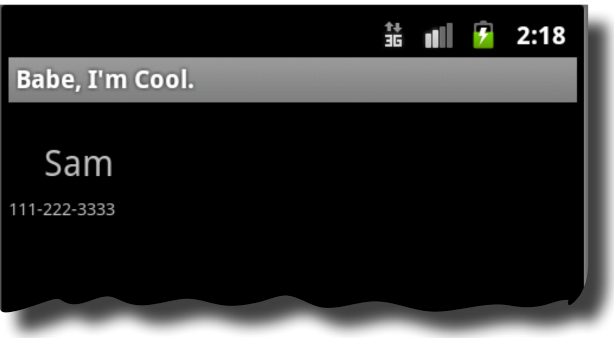
```
<TextView android:id="@+id/contact_phone"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:marginTop="5dp"      <- Add some vertical space.
        android:textSize="10dp"      <- Setting a smaller font...
        android:text="111-222-3333"
        android:layout_below="@+id/contact_name"
        />
```

main.xml



Test Drive

Now that the Phone Number view is positioned, run the app and make sure it's in the right place.



The phone number view is positioned under the contact name view, but all the way to the left of the screen.

How come the phone number field is all the way on the left?

... and so vertically close to the Contact Name view?

Align your views

You positioned the phone number TextView under the Contact Name TextView using the `android:layout_below` attribute. But why is it showing up all the way to the left?

In this case, positioning the phone number field below the Contact Name field controls the vertical position, but not the horizontal position. And since the horizontal position is not controlled it's defaulting to the left side.

You can use alignment properties to fine tune the position

When positioning isn't enough, you can use the layout alignment properties to position a View. There are attribute for aligning to the left, top, right, bottom, and baseline of another View.

android:layout_alignTop ← Align to the top

Align to the baseline. → **android:layout_alignBaseline**

android:layout_alignRight ← Align to the right.

Align to the left of another view. → **android:layout_alignLeft**

Align to the bottom → **android:layout_alignBottom**

Just like the `android:layout_below` attribute, pass the ID of the View you want to align to.

Align to the left... ... of the contact_name view.

```
android:layout_alignLeft  =  "@+id/contact_name"
```

Here's the complete layout so far

Adding bits and pieces at a time can make it hard to see the big picture. Take a minute and look at your complete layout so far.

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent">
<TextView android:id="@+id/contact_name"
        android:text="Sam"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:layout_alignParentLeft="true"
        android:layout_marginLeft="20dp"
        android:layout_marginTop="20dp"
        android:textSize="20dp"
        />
<TextView android:id="@+id/contact_phone"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="10dp"
        android:text="111-222-3333"
        android:layout_below="@+id/contact_name"
        android:layout_alignLeft="@+id/contact_name"
        android:layout_marginTop="5dp"
        />
</RelativeLayout>
```

RelativeLayout declaration.

Contact name TextView positioned to the top left side of the screen

The contact phone number TextView declaration. Right under the contact name TextView and aligned to the left to match the contact name horizontal position.

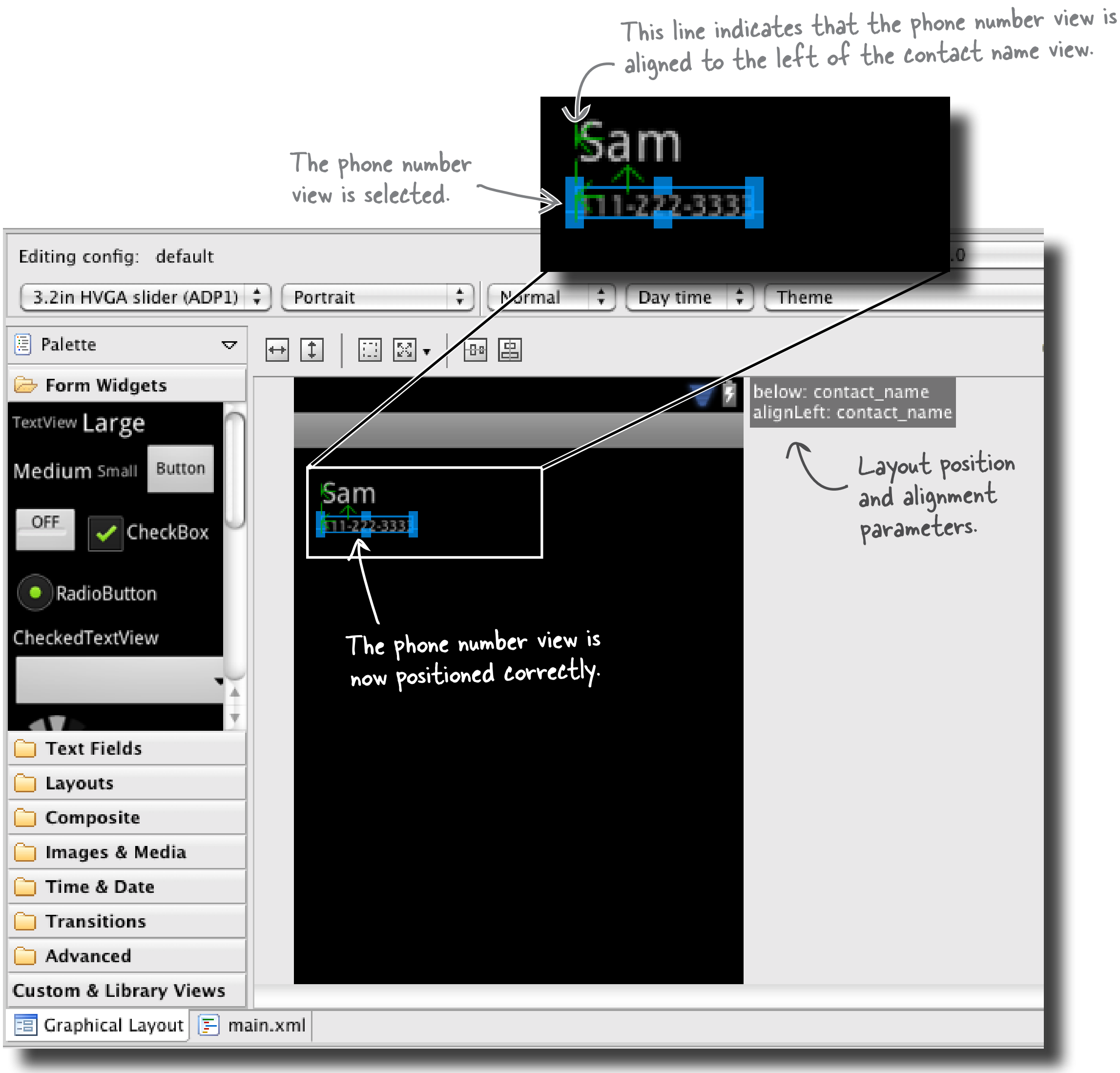
The attribute aligning this TextView to match the horizontal position of the contact name TextView.



main.xml

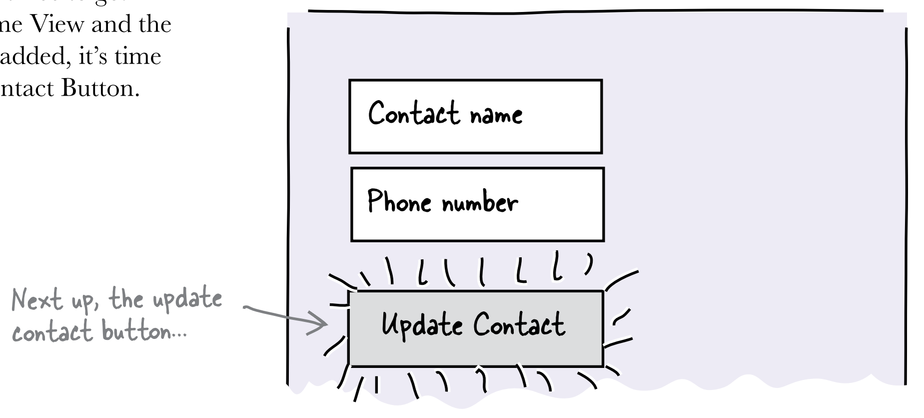
Test Drive

Now that you have both the contact name and phone number Views positioned in the layout, check and make sure your positioning worked correctly. But this time, instead of launching the app, just click on the **Graphical Layout** tab. Not only will you be able to see if your layout worked, but you can see graphical layout position and alignment indicators if you click on a View on the screen.



Add the Update Contact button

You've already positioned two Views on the screen and just three to go! With the Contact Name View and the Phone Number View added, it's time to add the Update Contact Button.



Sharpen your pencil

Below is the declaration of the update contact button. Position the `Button` below the phone number View and aligning to the left of the Contact Name View. Give it `10dp` of vertical margin.

```
<Button android:id="@+id/update_contact_button"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:text="Update Contact"

       ....................................................................

       ....................................................................

       ....................................................................
/>
```

Sharpen your pencil Solution

Below is the declaration of the update contact button. You should have positioned the `Button` below the phone number View and aligned it to the left of the Contact Name View. Give it `10dp` of vertical margin.

```
<Button android:id="@+id/update_contact"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Update Contact"

        android:layout_below="@+id/contact_phone"

        android:layout_alignLeft="@+id/contact_name"

        android:layout_marginTop="10dp"
/>
```

Margin top is vertical space.

there are no Dumb Questions

Q: Why is the button aligned to the left of the contact name view and not the contact phone view?

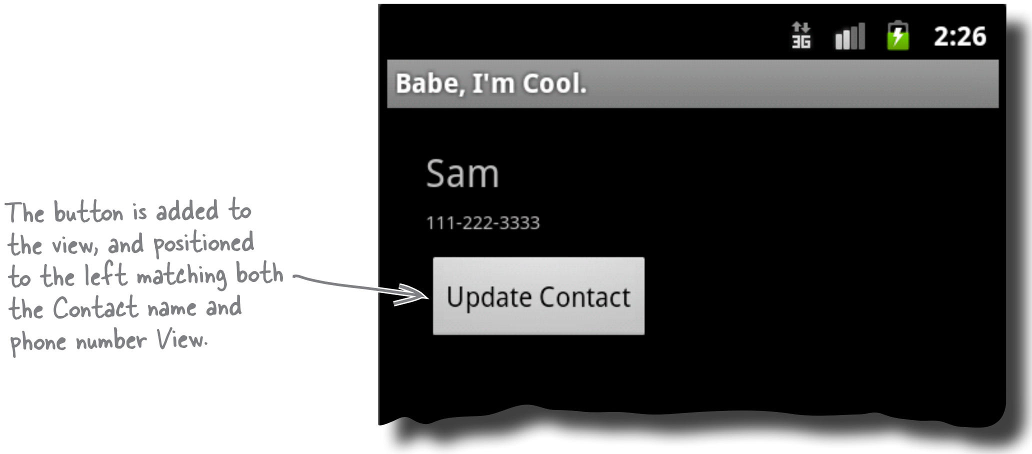
A: Either one would work. The reason is that the contact phone view i aligned to the left of the contact name view. So setting the button to align to the left of either the contact name or contact phone would both work. Sometimes it's better to have a single alignment view that is referenced by multiple views and other times is better to have the layout positioning and alignment refer to the save view. It's really up to you how you want to organize your layouts.

Q: What if I want to position a View relative to another View on the screen, but align with the parent? Can I mix and match like that?

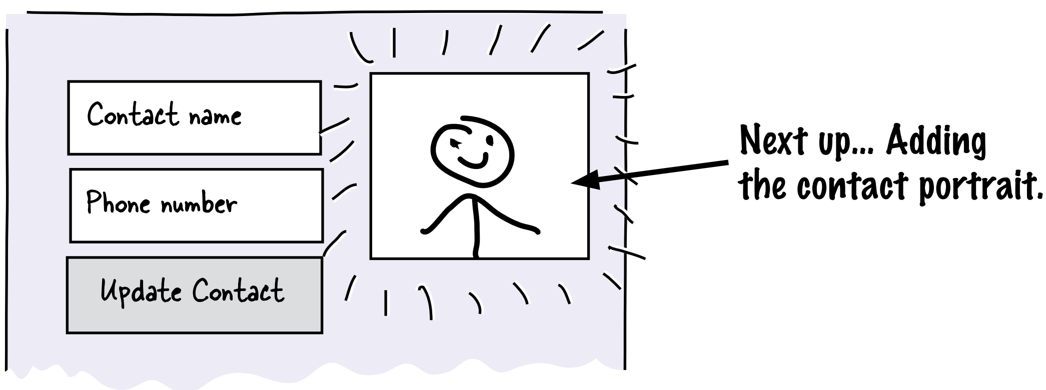
A: You sure can! Say you wanted to position the button below the phone number view but align it on the right side of the screen. You could use the `layoutBelow` attribute to position the button below, but use the `layout_alignParentRight` attribute to align it to the right side of the screen. Pretty slick!

Test Drive

It's a good idea to test your layouts early and often, especially when you're working with `RelativeLayout`! This way, you don't go too far down a path if a View isn't positioned correctly.



This is looking great!



Getting ready to add the contact portrait

Your ready to add the portrait `ImageView` to the screen. This `ImageView` is going to diaplay the avatar associated with the contact. You're going need an image to use to position it and make sure it looks OK.

For now. just set the background to an RGB color and give it a size in DPs. This way, you can layout the view on the screen and make sure it's positioned correctly.

Sharpen your pencil

Adjust the `ImageView` attributes below. Align it to the top of the Contact Name `TextView` and to the right hand side of the screen. Also, add `20dp` margin on the right to give the `ImageView` a border between it and the right edge of the screen.

```
<ImageView android:id="@+id/contact_portrait"
       android:layout_width="50dp"
       android:layout_height="50dp"
       android:background="#aaa"
       android:adjustViewBounds="true"

       ..................................................................

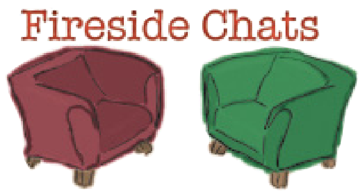
       ..................................................................

       ..................................................................

/>
```

Set the background to a light gray so you can easily see it to position it.

android:adjustViewBounds is set to true so the image will adjust as needed.



Tonight's talk: **Is Relative Layout The New GridBagLayout?**

RelativeLayout:

Shudder. I can't believe I'm here with GridBagLayout.

GridBagLayout:

Talk about getting off on the wrong foot! What's wrong with being here with me?

RelativeLayout:

For everyone out in the audience, GridBagLayout was the magical layout in Java's Swing desktop UI Toolkit that was supposed to be able to layout your whole screen in one layout.

GridBagLayout:

Yup. That sounds about accurate.

RelativeLayout:

Sure, except that you are impossible to use! You have made countless developers cry. Seriously!

GridBagLayout:

Now wait a minute, that's just unfair! It's true I have a rather complex grid structure that my developers have to learn, then place each component in the right position in the grid...

RelativeLayout:

Exactly! See, I have no grid. You just position a component somewhere on the screen and position other components around it. Simple!

GridBagLayout:

You know, I don't have to sit here and take this kind of badgering from you!

RelativeLayout:

I'm sorry! I didn't mean to offend you. I just wanted to point out that although we both can layout very complex sets of components we do so very differently. I use relative positioning to create very complex layouts...

GridBagLayout:

Right, and I use a grid.

RelativeLayout:

Exactly. But I'm just saying I'm waaaaaay easier to work with than you are.

GridBagLayout:

OK, sure. I *do* require a person willing to devote effort learning and working with me.

RelativeLayout:

Ha! There you have it. I'm easier to use!

GridBagLayout:

OK, you are easier to use. Are you happy now?

RelativeLayout:

Yes. Yes, I am.

Sharpen your pencil Solution

You should have adjust the `ImageView` attributes below, aligning it to the top of the Contact Name `TextView` and to the right hand side of the screen. You should have also added a `20dp` margin on the right to give the `ImageView` a border between it and the right edge of the screen.

```
<ImageView android:id="@+id/portrait"
       android:layout_width="50dp"
       android:layout_height="50dp"
       android:background="#aaa"
       android:adjustViewBounds="true"
       android:layout_alignParentRight="true"
       android:layout_alignTop="@+id/contact_name"
       android:layout_marginRight="20dp"
/>
```

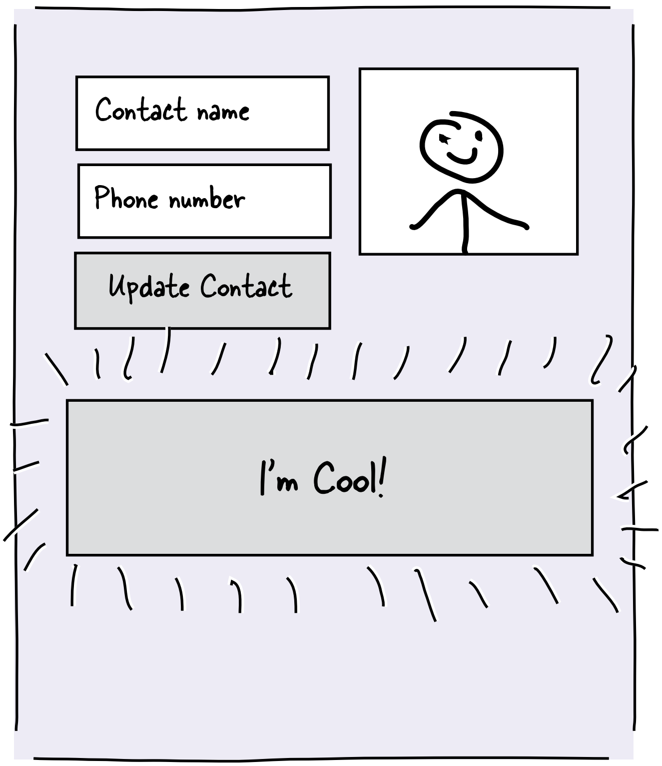
Align the view to the top right of the screen. (→ android:layout_alignParentRight="true")

Give a little space on the right. (→ android:layout_marginRight="20dp")



Time to add the "I'm Cool" button

OK, you've only got *one* more View to add to the screen... the big I'm Cool Button.



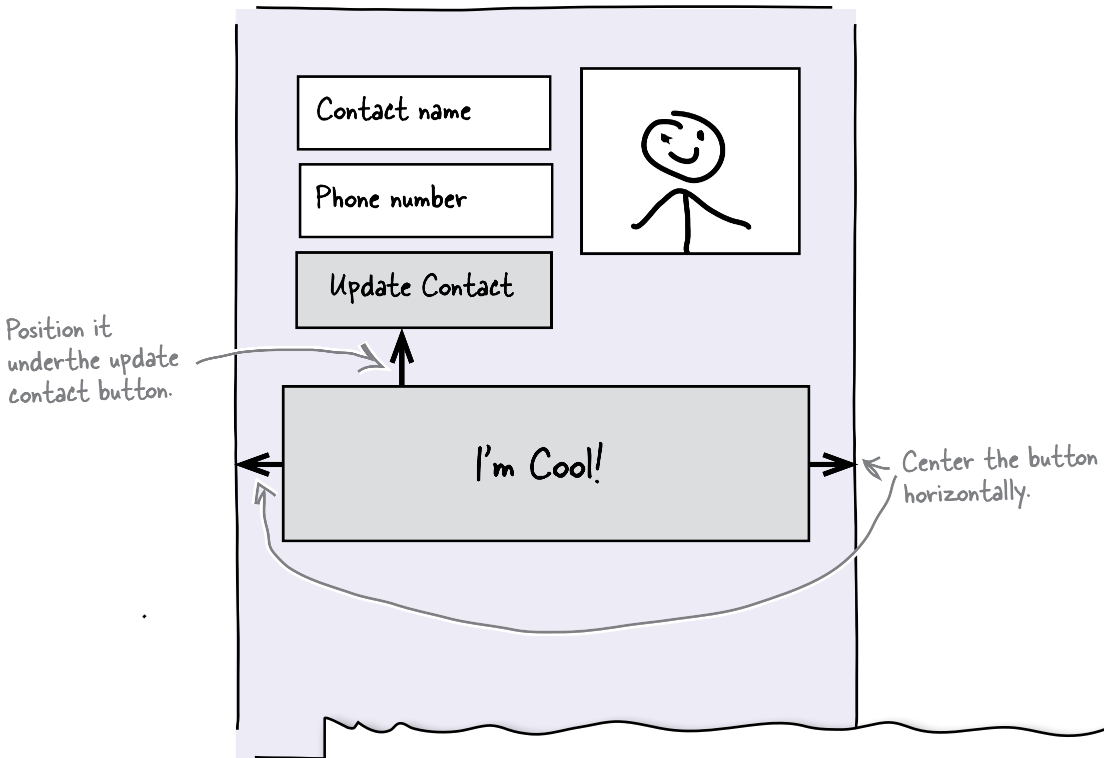


BRAIN POWER

How would you position the I'm Cool button?
What component would you align it with?
How are you going to position it?

Positioning the "I'm Cool" button

Did you think about how you could position the I'm Cool button? What did you come up with? One option you may have come up with is adding the button under the Update Contact and giving a little margin to the left.




```
<Button android:id="@+id/im_cool"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="I'm Cool!"
        android:layout_centerHorizontal="true"
        android:layout_below="@+id/update_contact"
        />
```



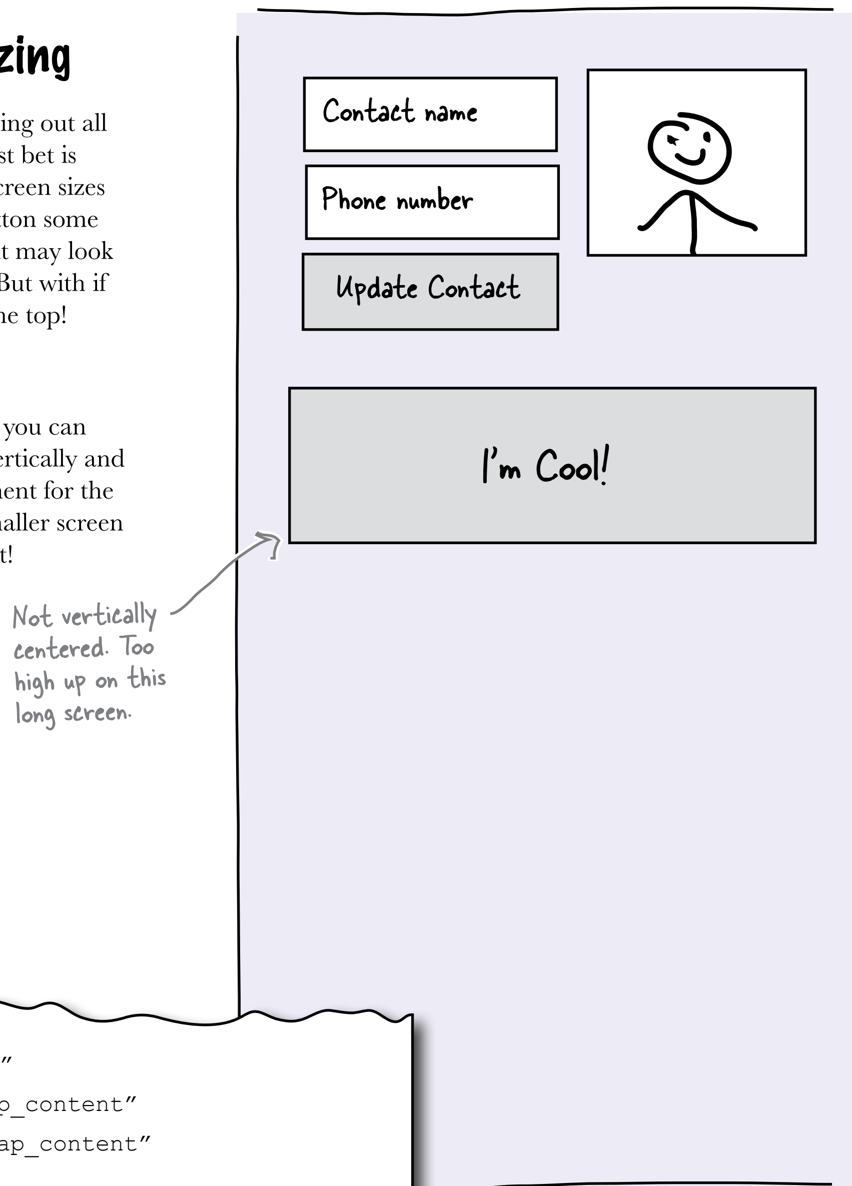
main.xml

Always think about resizing

The truth is, new Android deveices are coming out all the time with different screen sizes. Your best bet is think ahead and try and plan for as many screen sizes as possible. If you position the I'm Cool button some distance below the Update contact button, it may look good on some screens that your testing on. But with if the screen is really long? It'll be pinned to the top!

So what can I do?

There is another useful positioning element you can use to center the view in the parent- both vertically and horizontally. If you use that positioned element for the I'm Cool button, it would look ok on the smaller screen on the left AND the long screen on the right!




```
<Button android:id="@+id/im_cool"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="I'm Cool!"
        android:android:layout_centerInParent="true"
        />
```

main.xml

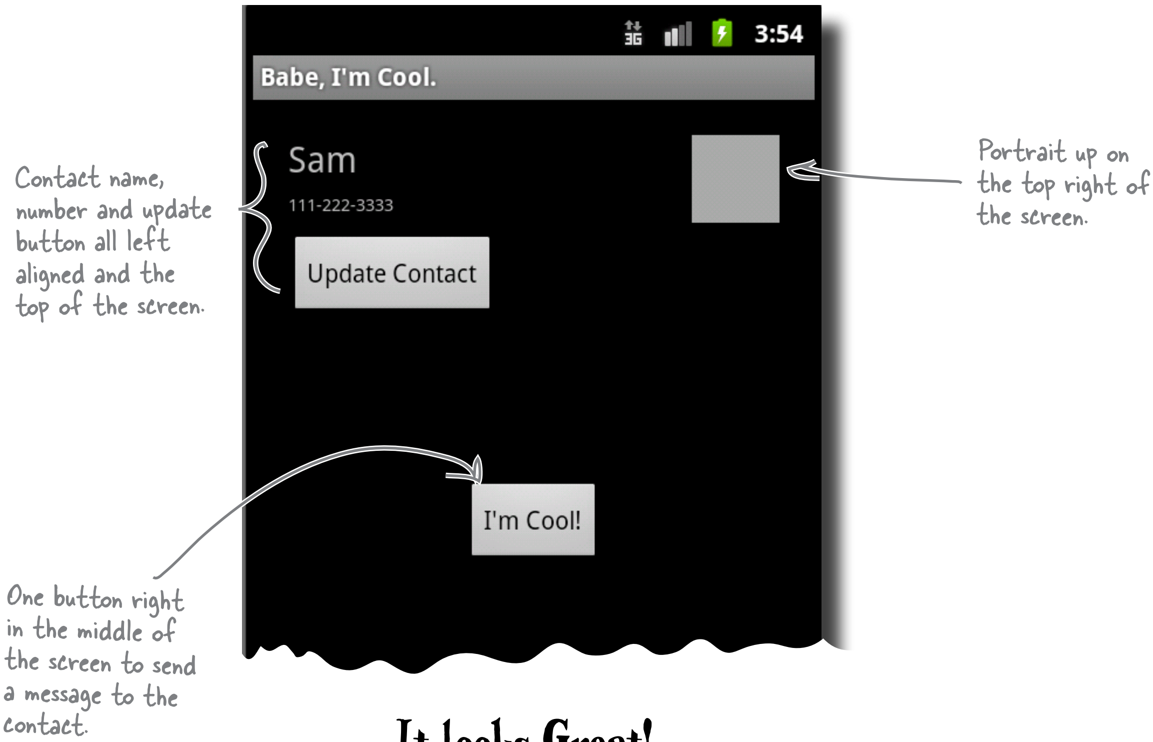
Center this View vertically and horizontally in the parent.

You're done! Now let's take a look at the completed layout.



Test Drive

All of the Views are laid out on the screen and (hopefully) positioned properly. Run the app in the emulator and make sure everything is where you expect it.



It looks Great!



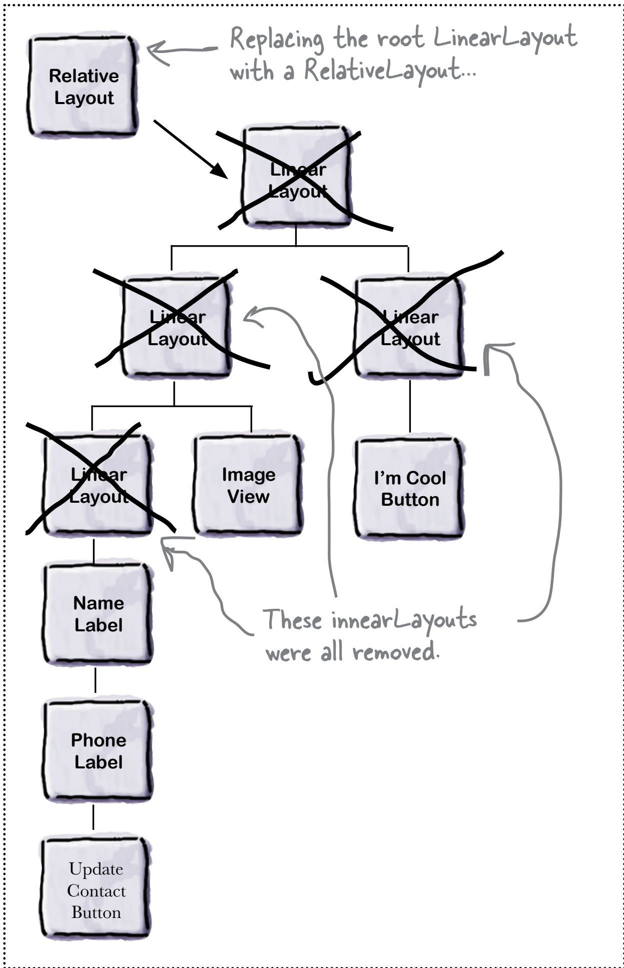
You may have noticed the I'm Cool butt is a little small.

The button is a little small now, you'll be fixing that in Chapter 12. There, you'll learn some advanced graphics techniques and make this button a large graphic.

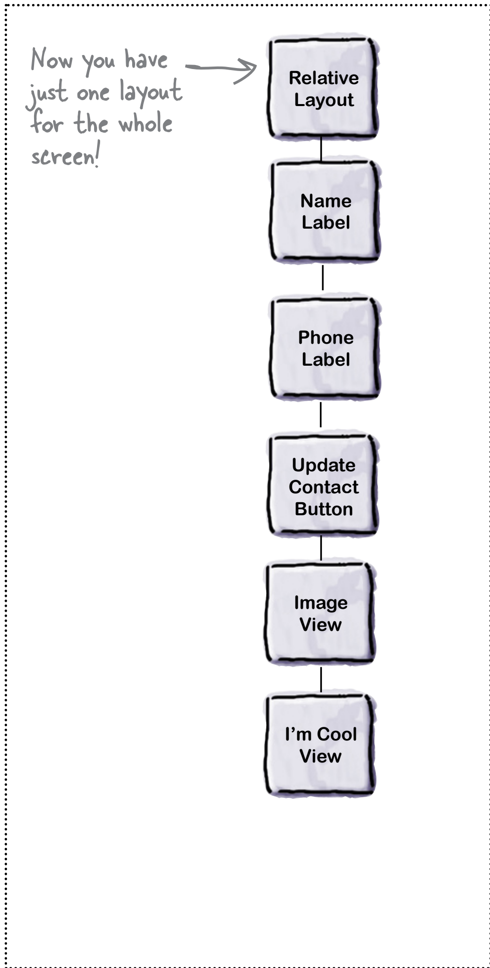
Comparing the layouts

With the screen layout all finished using `RelativeLayout`, let's go back and compare the tree of the nested `LinearLayouts` with the new and improved `RelativeLayout`.

Before



After



Not only is everything laid out correctly... but it's all done with one layout.



Go Off Piste

You're quickly becoming a `RealativeLayout` master by the end of this chapter. If you're ready for more, here are a few pointers to more information on `RealativeLayout` and other cool layouts.

RelativeLayout docs

Go to http://developer.android.com/reference/android/widget/RelativeLayout.html for detailed RelativeLayout documentation.

Other layouts

RelativeLayout isn't the only layout manager on the block. Go to http://developer.android.com/guide/topics/ui/layout-objects.html for a quick look at other layouts not covered here, including FrameLayout and TableLayout.

Write your own layout

Layouts are not magical bits of code passed down in the SDK, and if you're doing something special you can write your own! Check out the documentation for the ViewGroup abstract class (http://developer.android.com/reference/android/view/ViewGroup.html) for information on writing your own layouts.



Your Android Toolbox

You just laid out your first screen with RelativeLayout. Let's take a look at what you've learned.

RelativeLayout process

1. Add an anchored view aligned with the parent
2. Add more views relative to the others views added
3. Add more anchored views and views relative to other views as neeeded
4. Rinse and repeat!

BULLET POINTS

- Too many nested LinearLayout can slow down your application performance.
- Use RelativeLayout to optimize deeply nested LinearLayouts.
- Align views to the parent positions using `alignParentBotton`, `alignParentTop`, `alignParentRight`, and `alignParentLeft`.
- Layout Views relative to other on screen views using `layout_above`, `layout_below`, `layout_toRightOf`, and `layout_toLeftOf`.
- Align Views relative to other on screen views using `layout_alignTop`, `layout_alignRight`, `layout_aligntLeft`, `layout_aligntBaseline`, and `layout_alignBottom`.

11 content providers

Working with device contacts



One of the greatest things about Android is how well applications can work together. So far, you've built an apps that access content on the Web (like the NASA Daily Image app) and apps that generate their own content (like the TimeTracker app). But sometimes you need to access your users content on their device to make the app fit seamlessly into their user experience. Luckily, Android makes that super easy for you! In this chapter, you'll learn how to select contacts using contact selection built into the OS. You'll also learn how to query contacts stored on the device and a few different details about them.

Your app has a big problem at the moment...

... it doesn't actually work

Yet!

The app is looking good, but now I want it to be able to use it! Sam and I are heading out for a little bit. Check back with you later.

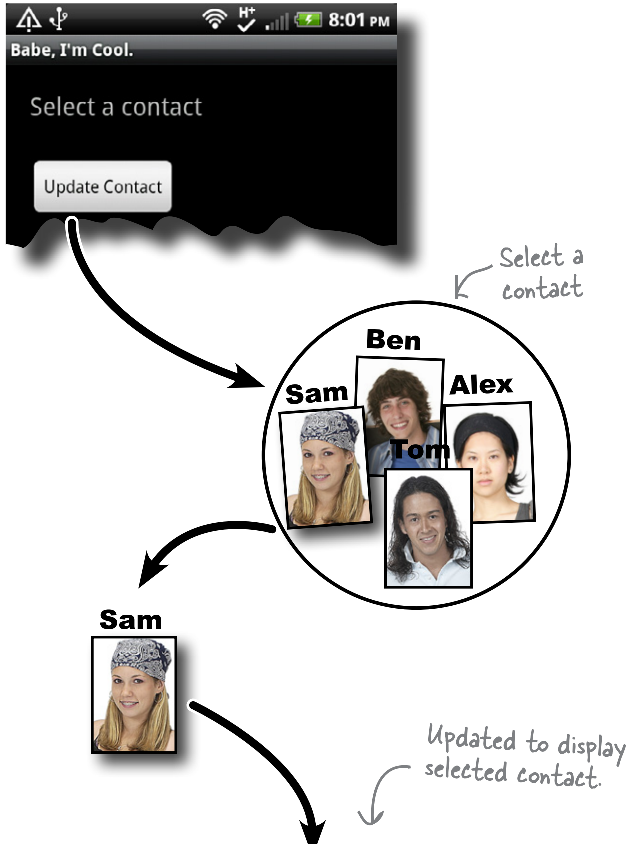


He's got a point, you know.

You just finished laying out all of the views, but that still won't allow Sam or Scott to send messages to each other. Let's get the guts of the app built out and get Sam and Scott messaging each by the time they get back.

Here's what you're going to do

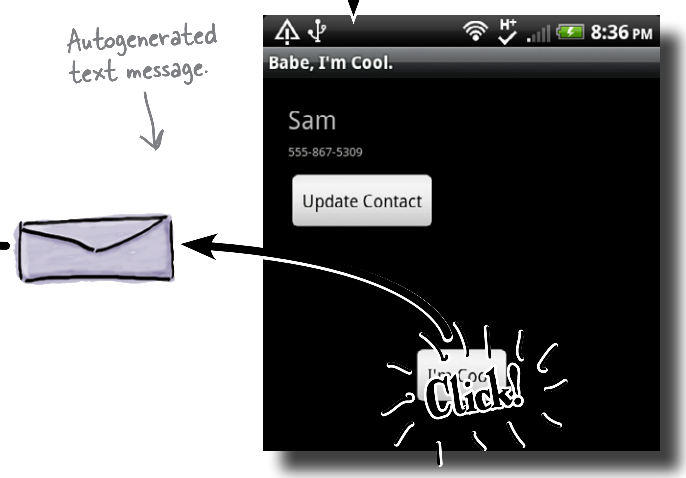
Sure, you have some work to do. The app doesn't have the functionality you need yet, but you laid the groundwork with the layout you built in the last chapter. Here is what you're going to do in ***this*** chapter to **make the app work**.



1. **Select a contact**
   Pressing the update contact button should show a screen allowing your users to select a contact from the phone. This way, your users won't have to enter contact information multiple times.

2. **Update the display**
   After the contact is selected, the contact display (the contact name, phone number and photo) should update to display the selected contact's information.

3. **Send a text message**
   This is the real user goal of the application. Once the contact is selected, your users should be able to press one button and have a text message automatically sent to their selected contact.





Make it clear that no contact is selected

When you first launch the app, *no contact is selected*. In the last chapter, you designed and constructed the user interface with some temporary contact information. But now that you're making the app work, start by making it clear that no contact is selected when it launches.

Start by adding a new method called `renderContact` and call it from `onCreate`. Right now this method will just display a message to select a contact. Later, it will display the contact you've selected.

```
public class ImCool extends Activity {

    public void onCreate(Bundle savedInstance) {
        super.onCreate(savedInstance);
        setContentView(R.layout.main);
        renderContact();
    }

    private void renderContact() {
        TextView contactNameView =
            (TextView) findViewById(R.id.contact_name);
        TextView contactPhoneView =
            (TextView) findViewById(R.id.contact_phone);
        ImageView photoView =
            (ImageView)findViewById(R.id.contact_photo);

        contactNameView.setText("Select a contact");
        contactPhoneView.setText("");
        photoView.setImageBitmap(null);
    }
}
```

Add a renderContact() method called from onCreate(). Right now this is just showing the "no contact" message but eventually this will display the contact iif there is one.

Get references to the contact display views.

Display a message in the name view and blank out the rest.



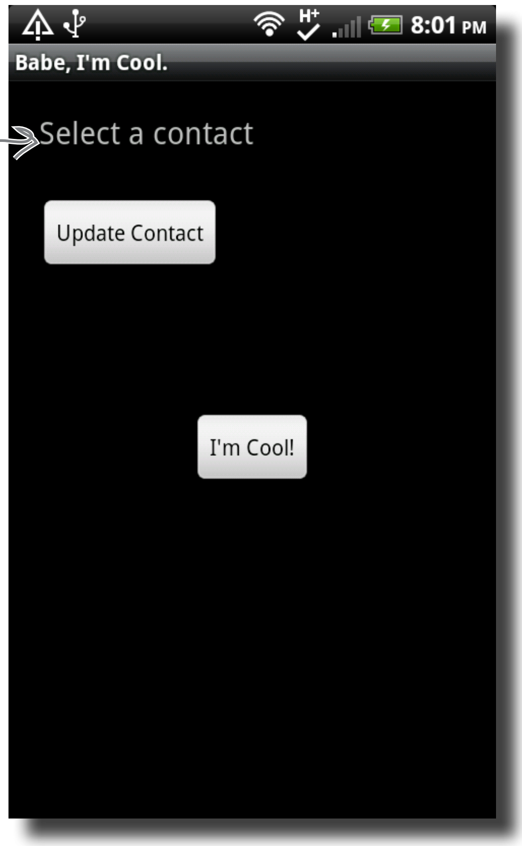
ImCool.java



Test Drive

Run the app now and verify that the “Select a contact” text appears in the contact name View.

Here is the “Select a contact” text displaying in the displayName field.



The render contact change looks good!

Now let’s get started selecting a contact...

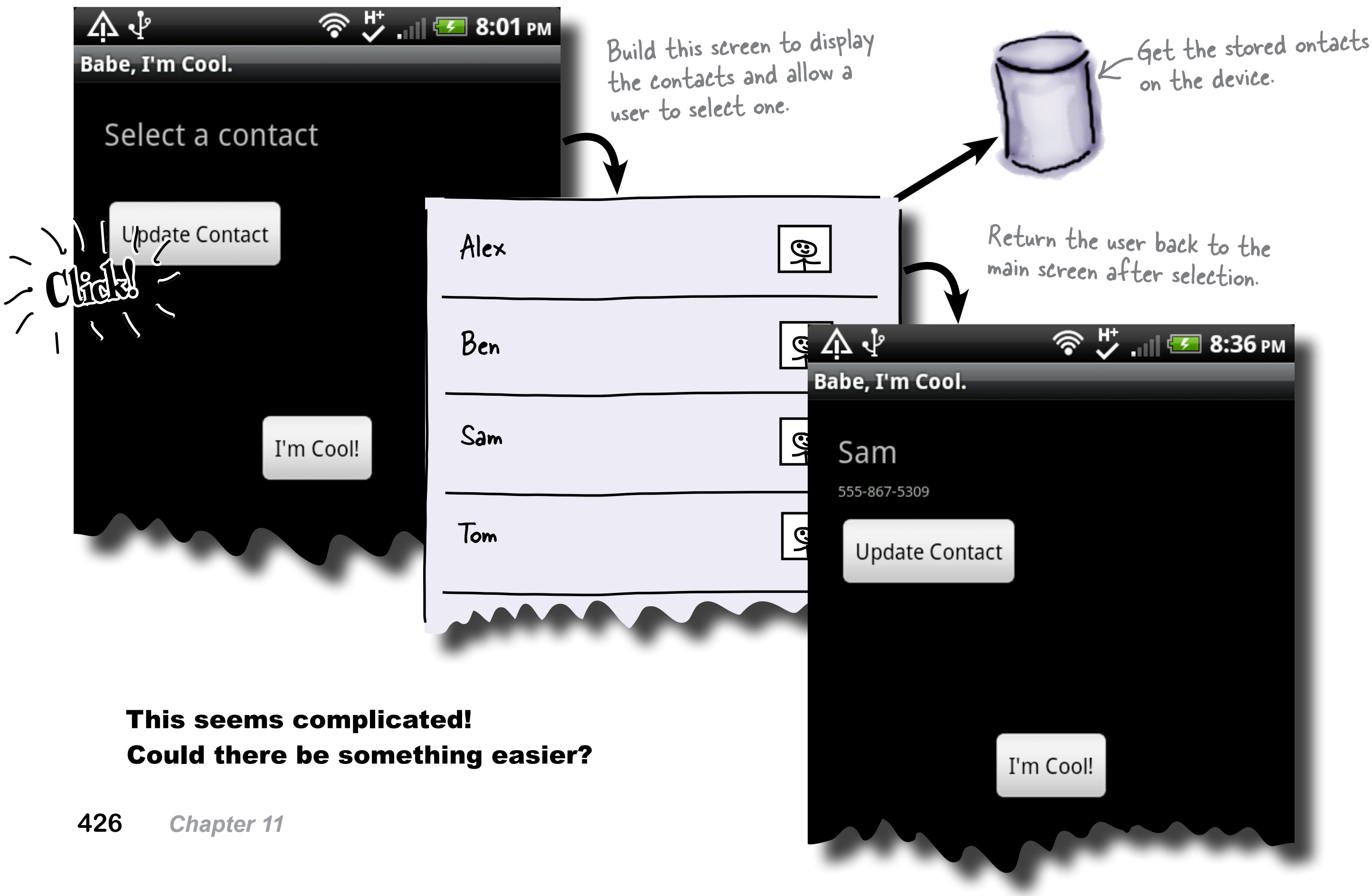
How do I select a contact?

You're ready to select a contact now. You ***could*** have them enter their contact's name and phone number and select a picture to make the app work. **But they've already entered that information into their phone, in their contact list.** So just let your users *select a contact from their contact list* and you'll save them a lot of boring data entry, and leave your app focused on the cool stuff.

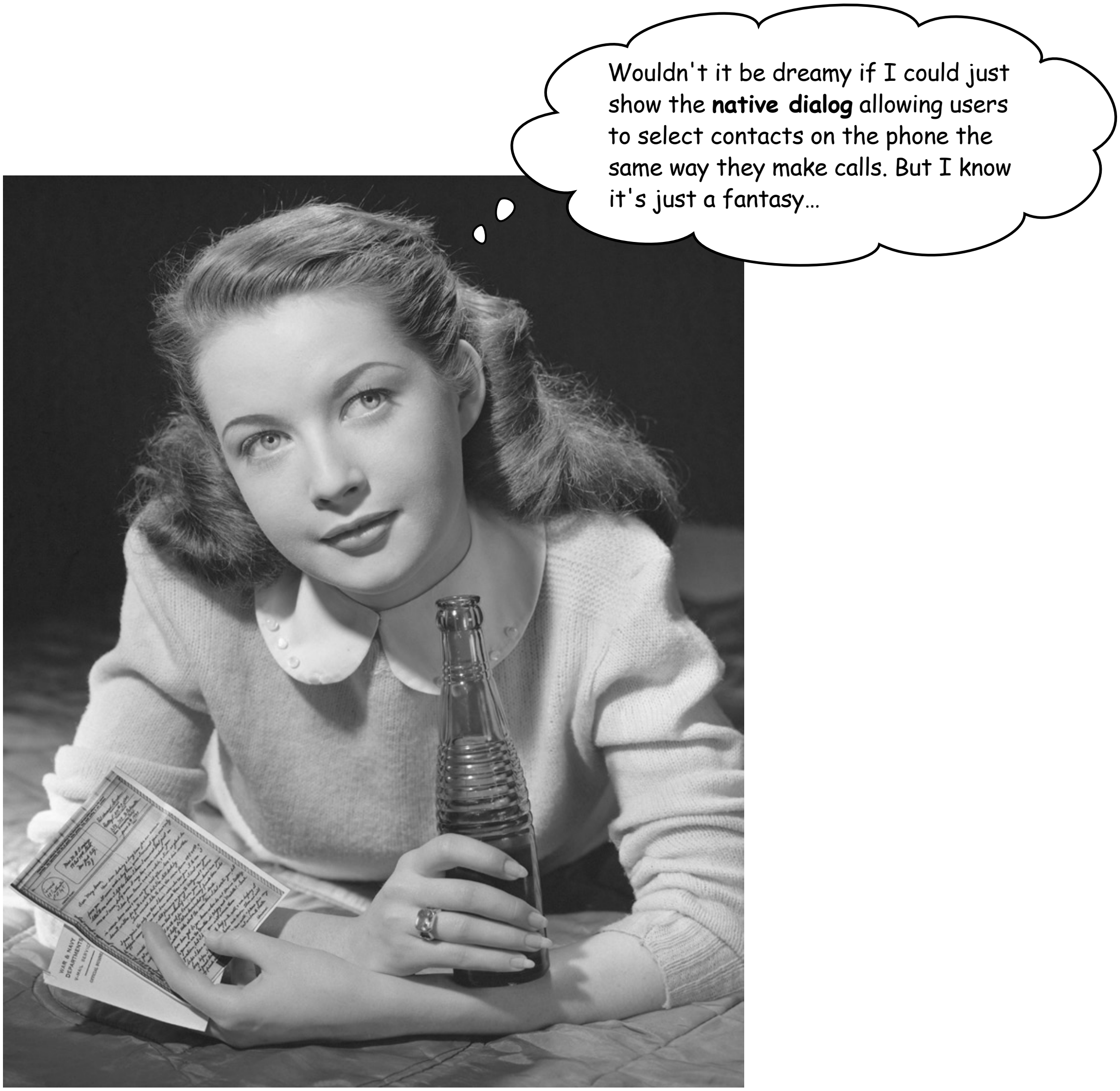
But how should you build a contact list selecting screen?

You could build a screen that loads and displays contacts stored on the phone...

Here is what the flow would look like if you built your own contact screen. When you press update contact, you'd go to your new screen and back to the main screen after you selected a contact.



This seems complicated! Could there be something easier?


Wouldn't it be dreamy if I could just show the **native dialog** allowing users to select contacts on the phone the same way they make calls. But I know it's just a fantasy...

Don't custom build...

Android already has behavior built in to select contacts. This is used to select contacts for phone calls and other native apps. But it can also be used by apps like yours so you don't have top build it yourself.

... use the native contact select screen

Using the native contact selection screen will keep the same flow, but you won't have to build it yourself.



there are no Dumb Questions

Q: Why is it better to use the native contact selection?

A: First of all you don't have to build it! But more importantly, it guarantees your users experience is the same as the native experience. If there is a modified version of the contact selection on your users' devices, they'll see whatever is native when you invoke the selection request. Also, if the native contact selection changes over time, you'll get whatever the latest behavior is automatically. If you built it yourself, it might look different than what your users are expecting.

Q: OK, I get that. But what if I really want to make a custom replacement for native behavior in my app?

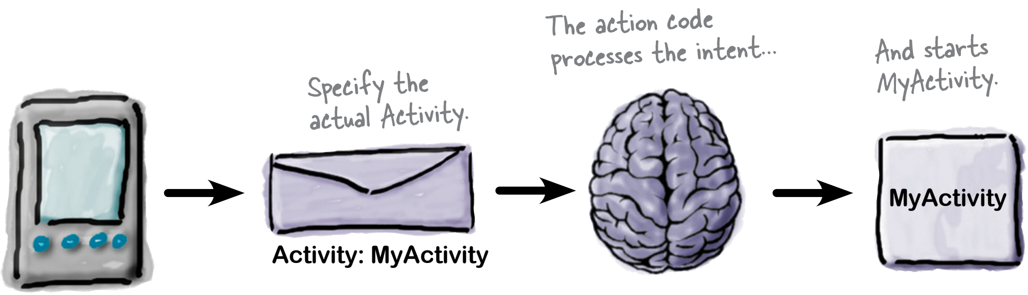
A: You could do that too. You could query the contact store directly and build a custom screen or component displaying the content and allowing your users to select contacts that way. But this chapter is going to focus on using the native selection.

Invoking the contact screen

OK, so it looks like the built in contact selection is the way to go. But how do you invoke it from the app?

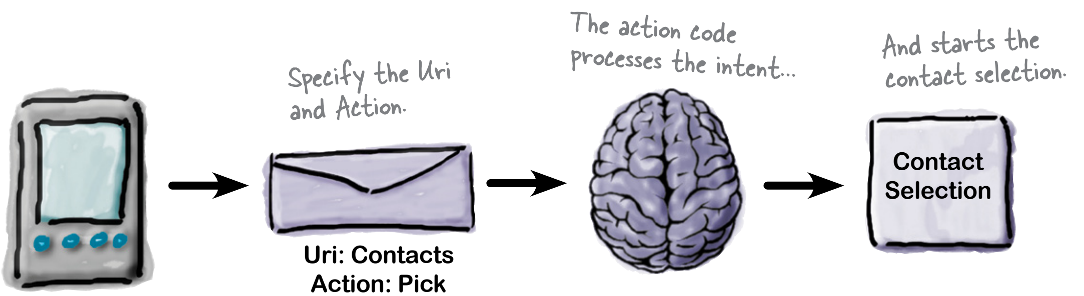
You can use Intents

Intents are a generic mechanism for invoking an action that the system can respond to. When you built the screen navigation in the TimeTracker app, you specified the Activity you wanted to invoke in the Intent. When the Android action code processed that Intent, it saw the reference to the Activity and invoked it directly.



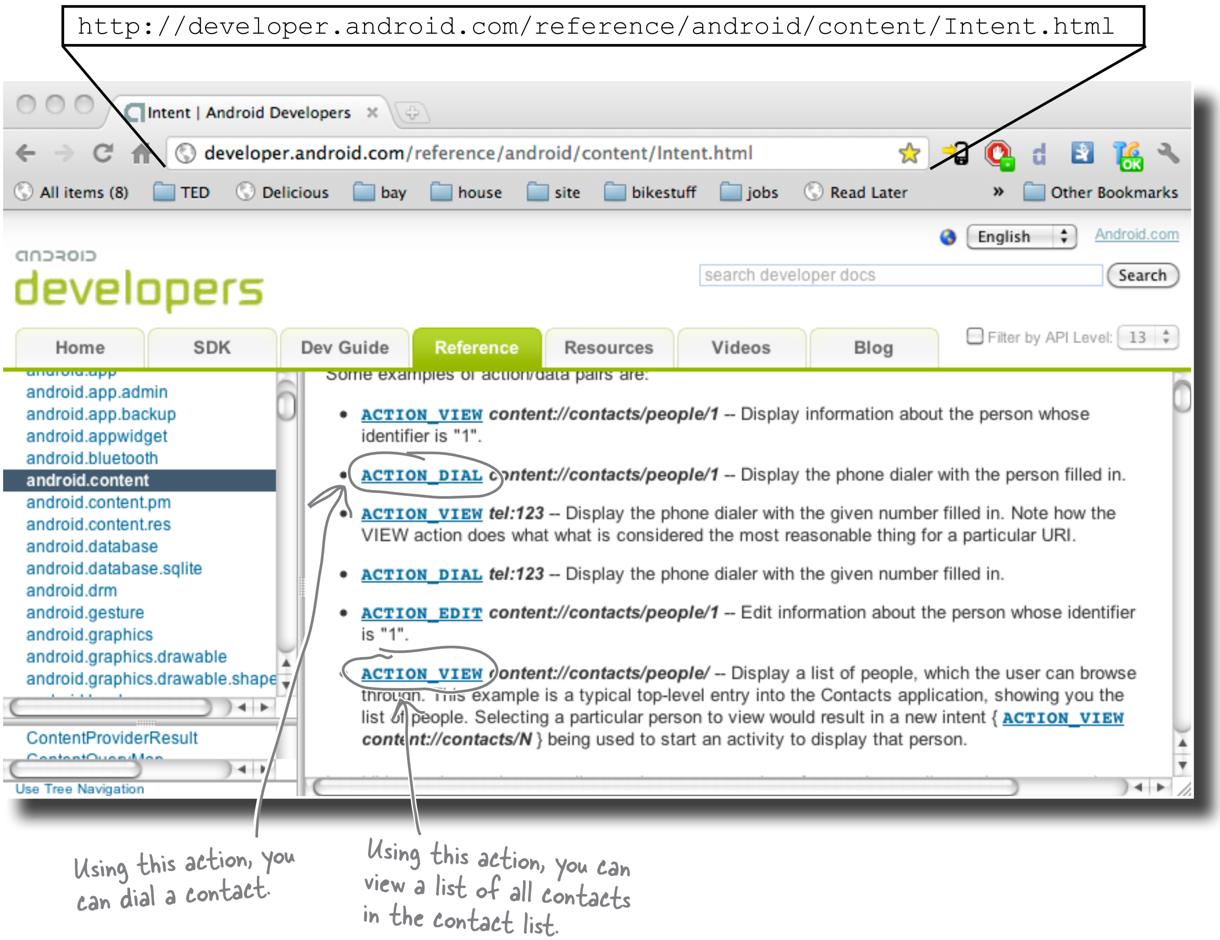
But you can be WAY more abstract than that.

You don't actually have to include a reference to an Activity in an Intent. You can also supply a Uri or a combination of Uri and an Action. And if you invoke the Intent, the Android action code looks for an Activity that responds to that Uri and invokes it.



Select the URL and Action

You can find extensive documentation for the Uris and Actions you can pass into an Intent in the Intent's online documentation. Go to `http://developer.android.com/reference/android/content/Intent.html` to take a closer look.



Creating an Intent

You need to create an `Intent` to select a contact from the contact list. You can create this Intent using the constructor that takes an `Action` and `Uri`.

The action that the activity will invoke.

The uri defining the data for the action.

```
Intent( String action, Uri uri );
```

The `Uri` is a reference to data on the device, while the Action says what to do with the data. So you'll pass in the `Intent.ACTION_PICK` constant. **But what about the Uri?**

`Uris` are actually human readable descriptions of where to find the data. The `Uri` to find all of the contacts in the phone's contact list is `content://com.android.contacts/contacts`.

But to make the types work with the constructor, you need to convert the string in a `Uri` object which you can do using `Uri.parse`.

```
Uri contactUri = Uri.parse(
      "content://com.android.contacts/contacts"
);
```

Hrm. You have a constant for the action but not the Uri. You sure there's not a constant for that too?



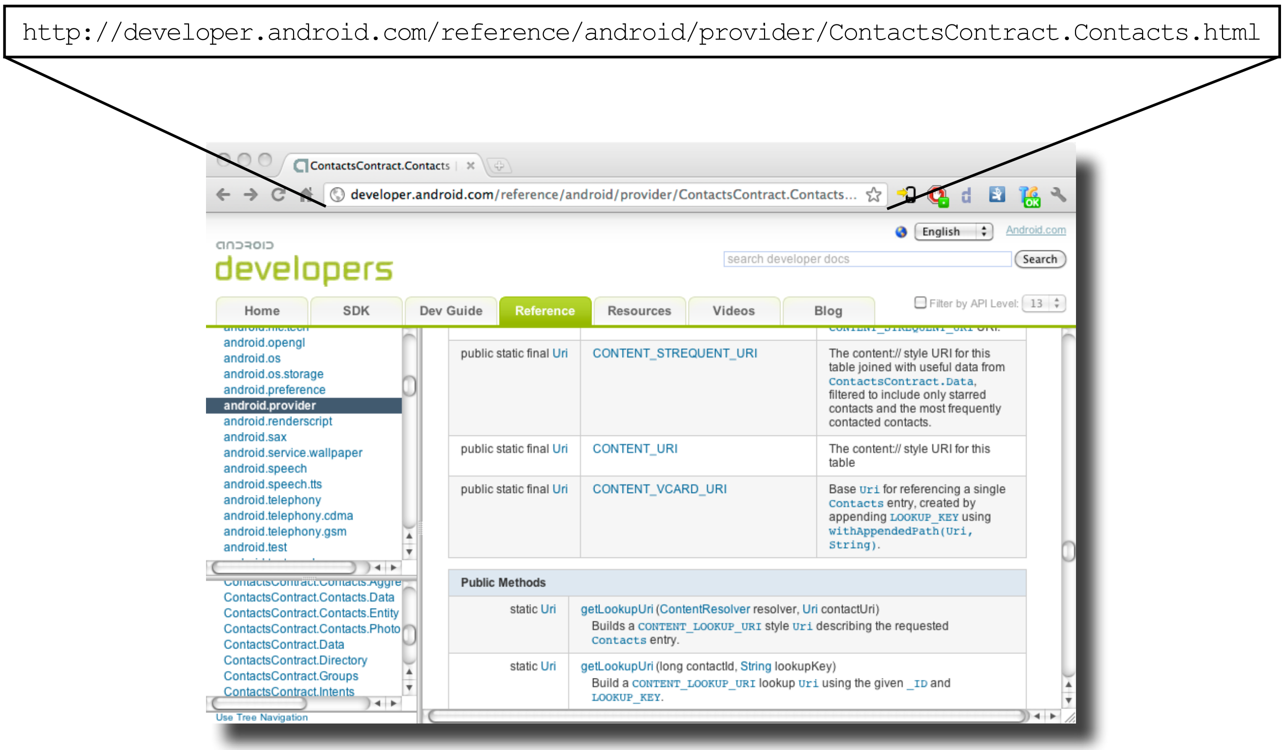
Is it a good idea to use a String to create the Uri or is there a constant you can use.

Let's take a look...

Use constants when you can

The `Uri` created by parsing the string will work, but raw strings are just a hassle to keep in your codebase. The format could change in the future or you could just have a typo in your code that the compiler wouldn't catch. Always best to use constants if you can. And there is just such a constant you can use.

Take a look at ContactsContract.Contacts



Get ready to invoke the new Intent

You could launch the new Activity by calling `startActivity`,. But in this case, you want to have the selected contact returned after the contact selection is complete. That's OK though, you can just use `startActivityForResult` just like when you built your own screens.



Contact Selection Intent Magnets

Below is the code for `onUpdateContact` and `onActivityResult`. Complete `startActivityForResult` by creating an Intent and passing in the `Action` and `Uri`. In `onActivityResult`, print the returned Intent to the Log to see what comes back.

```
private static final int PICK_CONTACT_REQUEST = 0;
```

Add a constant for a request type passed to startActivityForResult and verified on return on onActivityResult.

```
public void onUpdateContact(View view) {
        startActivityForResult(



              PICK_CONTACT_REQUEST
        );
}
```

Create the intent here with the action and the Uri.

```
protected void onActivityResult(int requestCode, int resultCode, Intent intent) {
        if (requestCode == PICK_CONTACT_REQUEST) {
              if (resultCode == RESULT_OK) {


              }
        }
}
```

Print a message printing out the returned intent to the log.



ImCool.java

Your magnets.

```
new Intent (
,
);
Log.d("Selection", intent.toString());
Intent.ACTION_PICK
ContactsContract.Contacts.CONTENT_URI
```



Contact Selection Intent Magnets Solution

Below is the code for `onUpdateContact` and `onActivityResult`. You should have completed `startActivityForResult` by creating an Intent and passing in the `Action` and `Uri`. In `onActivityResult`, you should have printed the returned Intent to the Log to see what comes back.

```
private static final int PICK_CONTACT_REQUEST = 0;
```

```
public void onUpdateContact(View view) {
    startActivityForResult(
        new Intent (
            Intent.ACTION_PICK  ,  ContactsContract.Contacts.CONTENT_URI
        );
        PICK_CONTACT_REQUEST
    );
}
```

Instantiate a new Intent.

Pass in the action to pick a contact.

Pass in the Uri for all contacts.

```
protected void onActivityResult(int requestCode, int resultCode, Intent intent) {
    if (requestCode == PICK_CONTACT_REQUEST) {
        if (resultCode == RESULT_OK) {
            Log.d("Selection", intent.toString());
        }
    }
}
```

Print the intent to the log so you can see what's passed back.



ImCool.java

Getting ready to test contact selection

The contact selection code is all ready to go, but there are a couple of things to update in your project before you run it. First you need to add the `onClick` property to the **Update Contact** button on the screen to invoke the `onUpdateContact` method.

```
<Button android:id="@+id/update_contact"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Update Contact"
        android:layout_below="@+id/contact_phone"
        android:layout_alignLeft="@+id/contact_name"
        android:layout_marginTop="10dp"
        android:onClick="onUpdateContact"
        />
```

Add an onClick property pointing to the onUpdateContact method you just wrote.



main.xml

Now add the **`READ_CONTACTS`** permission to your **`AndroidManifest.xml`** file. Without it, you'll get an error when you try and access the contacts in your app. After all, a users contacts are sensitive information so you need to ask and they need to give you permission. **This should also clue you in to being really sensitive to what you do with that access.**

```
    </application>
    <uses-sdk android:minSdkVersion="10" android:targetSdkVersion="10" />
    <uses-permission android:name="android.permission.READ_CONTACTS"/>

</manifest>
```

Permission to access the contacts stored on the device.

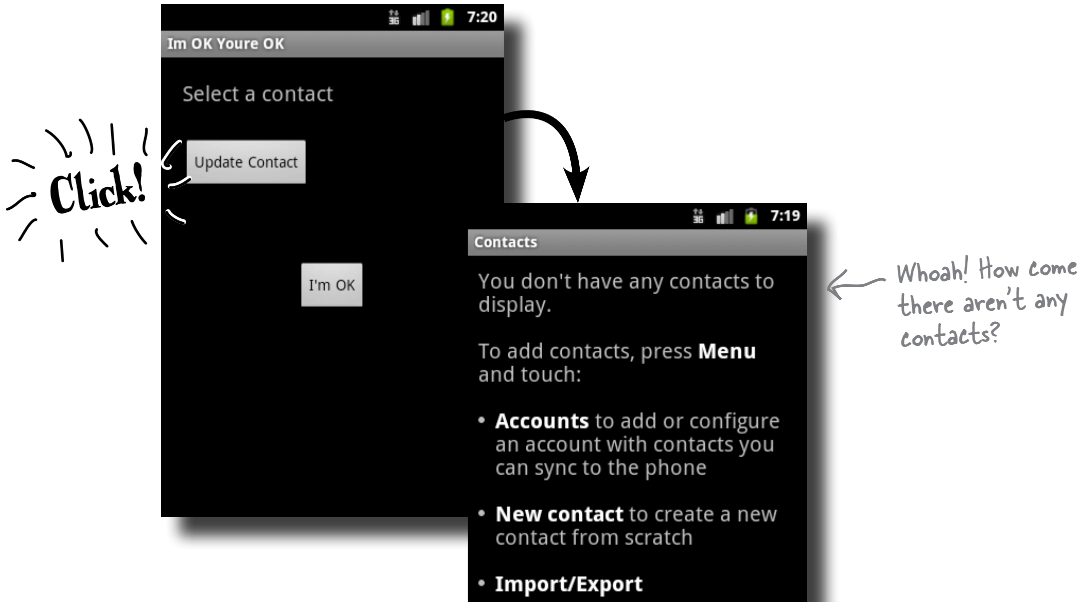


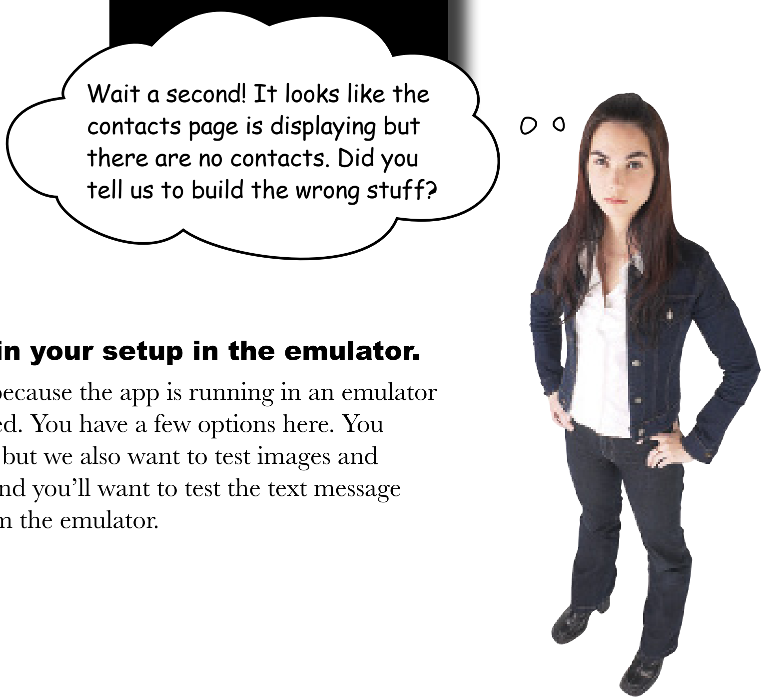
AndroidManifest.xml



Test Drive

Now that the Intent is created and being started, you should see the contact lost display when you press the Update Contact button. Go ahead and run the app and check to make sure it's working.





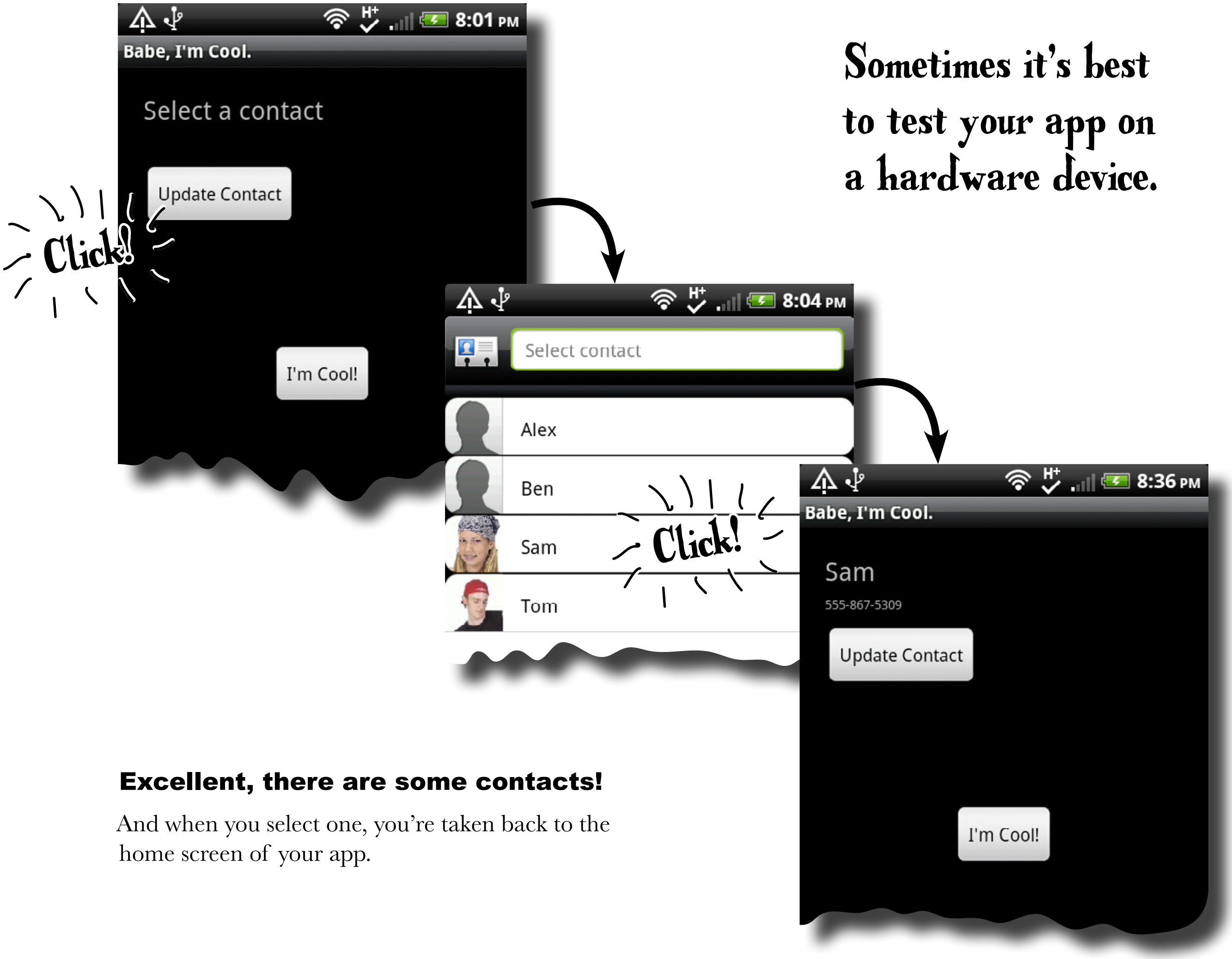
You don't have any contacts in your setup in the emulator.

The reason you're seeing this screen is because the app is running in an emulator that doesn't have any contacts configured. You have a few options here. You could create the contacts on the phone, but we also want to test images and images are hard to test on the phone. And you'll want to test the text message sending anyway which you can't do from the emulator.

Run the app on a device

Plug in your Android device using USB and remember to turn on the option to allow non-market apps. Then just run the app again from Eclipse and select your hardware device.

Now that you're running the app on your device, when you go to the Contact Selection screen, you should see a populated list of contacts. Click on a contact and you'll be taken back to the home screen.



Sometimes it's best to test your app on a hardware device.

Excellent, there are some contacts!

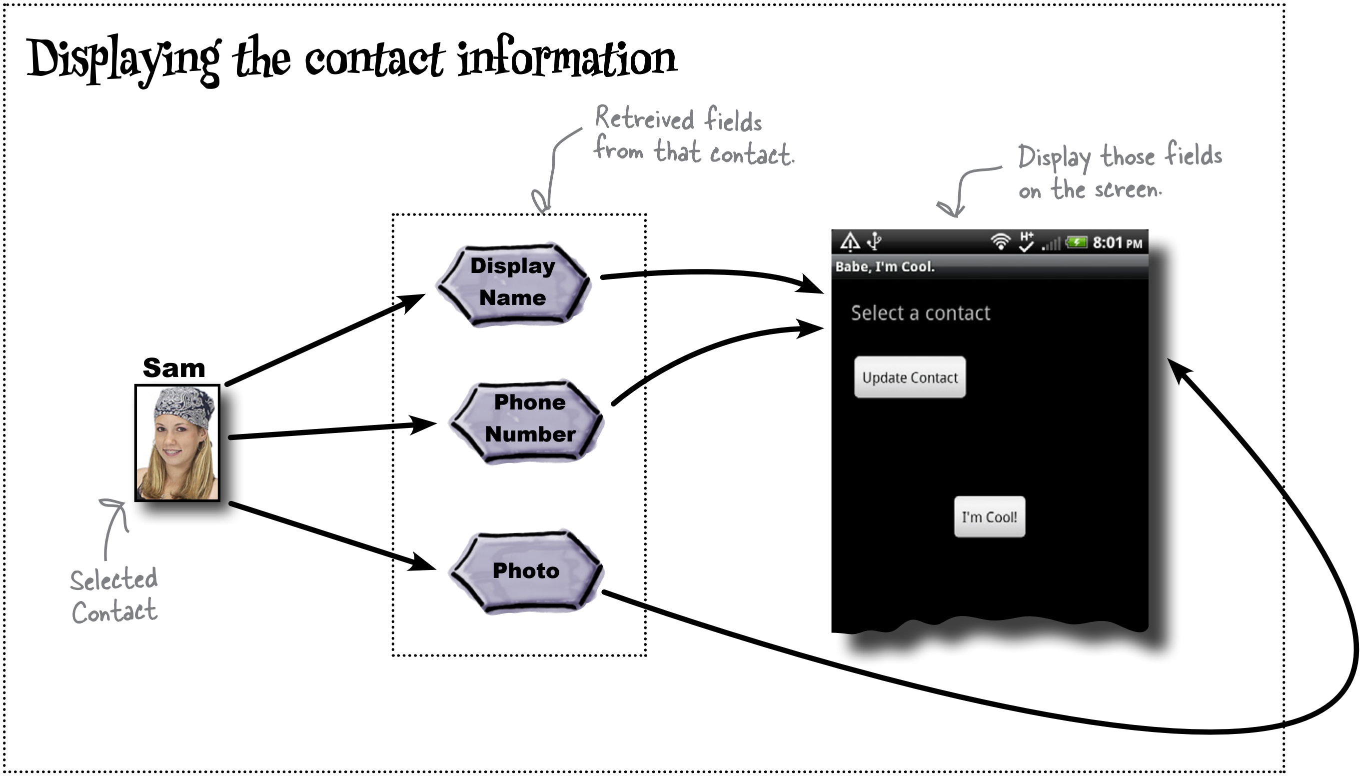
And when you select one, you're taken back to the home screen of your app.



Definitely! Displaying the sected contact is next on the list.

Now that the contact is being selected, it's time to display that contact on the home screen. To get this working, you'll need to get a reference to the contact that was selected, retrieve the display name, phone number and photo for that contact and display it on the screen.

Displaying the contact information



Start by looking at what's coming back

You're already getting the contact back to the ImCool Activity in `onActivityResult`. You also put a log statement in there to see what the returned `Intent` contains for its data. Take a quick look at the log and see what came back. You should see one line that looks something like this:

LogCat logging printout

This string starting with "content://" is the data returned from the contact selection.

```
08-10 15:44:52.131: DEBUG/Intent Data(355): content://com.android.contacts/contacts/
lookup/0r1-512D45/1
```

What is that string referencing?

If you're thinking that the string printing out in the logs looks kind of like some kind of a local web address, you're not too far off. It's actually a **URI**, or ***Uniform Resource Identifier*** which is a string that locates a specific resource. The different between the URI here and a web URL is that the **URI** here is an address for a local resource. In this case, the **URI** is a reference to the selected contact

Action to perform on the contacts.

The Uri portion pointing to contacts

Lookup Key for the selected contact.

```
content://com.android.contacts/contacts/lookup/0r1-512D45/1
```

Last known position for selected contact (a search optimization).

OK, so this **points to a contact**. But I don't want a reference to the contact, I want **the actual contact**!



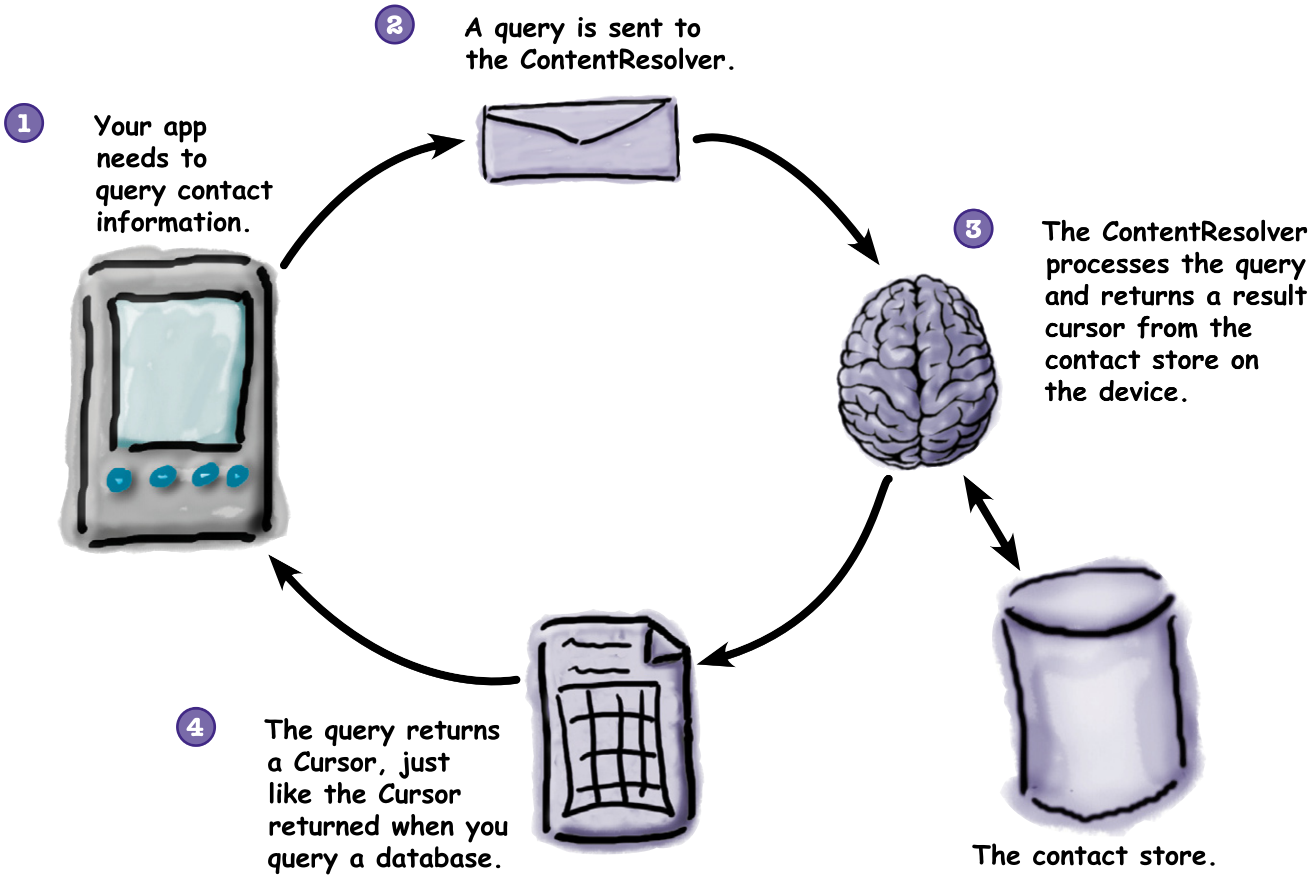
You can look up the contact using the URI

This `Uri` doesn't contain the real contact (which you need to get the name, phone number and photo to display on the home screen. But it does represent **a direct lookup to that contact**.

Accessing the contact

There is a contact data store built into every Android device. You can query the contact data store for specific contact information, like determining which contact has an associated phone number for building caller ID functionality, or in our case, just finding more properties for a contact that you already know about.

There is a utility class called `ContentResolver` that you can use to **query** the contacts. Using this `Uri` and a query to the `ContentResolver` , you can get to the raw contact! Then using what you learned when you iterated through database results, you'll iterate through the contact result `Cursor` it returns.



Update the code to use a URI

The `renderContact` method is currently hard coded to display the **no contact selected** state. But you're about to start populating the selected contact, so let's make it clear when there ***is*** and when there ***is not*** a selected contact. Then you can start filling in the code ***when a contact is selected***.

Update the `renderContact` method to pass in the `Uri`. If the `Uri` is null (meaning no contact is selected) then set the name, phone, and photo view to display the no contact selected state you setup at the beginning of the chapter. Also update the `onCreate` to call `renderContact` with a null `Uri` (since no `Uri` is selected) and from `onActivityResult` pass in the `Uri`.

Do this!

> Update `renderContact` to take a `Uri`. If that `Uri` is null, display the no contact selected state with the message to select a contact. Also update `onCreate` to pass in a null `Uri` and `onActivityResult` to pass in the `Uri` from the `Intent`.

Pass in the URI

```java
private void renderContact(Uri uri) {
    TextView contactNameView = (TextView) findViewById(R.id.contact_name);
    TextView contactPhoneView = (TextView) findViewById(R.id.contact_phone);
    ImageView contactPhotoView = (ImageView)findViewById(R.id.contact_photo);

    if (uri == null) {
        contactNameView.setText("Select a contact");
        contactPhoneView.setText("");
        contactPhotoView.setImageBitmap(null);
    } else {
        contactNameView.setText(getDisplayName(uri));
        contactPhoneView.setText(getMobileNumber(uri));
        contactPhotoView.setImageBitmap(getPhoto(uri));
    }
}
```

Check for a null URI. If null, there must be no contact.

Create helper methods for each data field you want to set on the screen.

ImCool.java

Then pass in a null URI in onCreate (since there is no contact selected yet)

```
public void onCreate(Bundle savedInstance) {
        super.onCreate(savedInstance);
        setContentView(R.layout.main);
        renderContact(null);
}
```

Pass in a null Uri onCreate since no contact has been selected yet

And in onActivityResult, pass the URI to renderContact.

```
protected void onActivityResult(int requestCode, int resultCode, Intent intent) {
        if (requestCode == PICK_CONTACT_REQUEST) {
                if (resultCode == RESULT_OK) {
                        renderContact(intent.getData());
                }
        }
}
```

Pass the Uri (the data from the intent) on to renderContact.

Finally, create stub methods for the three display methods. You'll be implementing these yourself!

Stub method for retreiving the display values for a contact.

This method will return the display name for the contact.

```
private String getDisplayName(Uri uri) { return null; }
```

This will return the MOBILE number for the contact.

```
private String getMobileNumber(Uri uri) { return null; }
```

This last method will return the photo for the contact.

```
private String getPhoto(Uri uri) { return null; }
```

Start with the display name

With that bit of code reorganization, you now have three contact detail methods to implement and the contact display will be up and running. Let's start with `getDisplayName`.

The three contact detail access methods you're going to implement.

Start with implementing getDisplayName.

```
private String getDisplayName(Uri uri) { return null; }

private String getMobileNumber(Uri uri) { return null; }

private String getPhoto(Uri uri) { return null; }
```



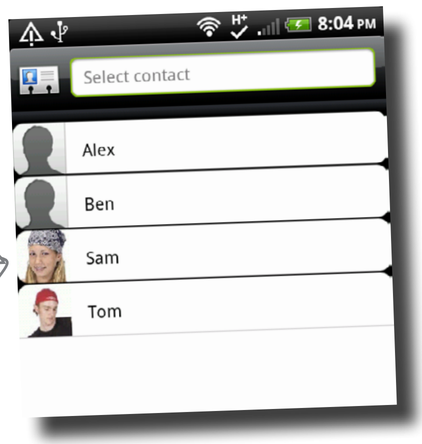
ImCool.java

So what does this method need to do?

This method needs to retrieve the name of the contact. The display name is the name that displayed in the list of contacts that you selected. Scott selected Sam from his contact list, so this method should return "Sam" to display it on the home screen. This way Scott will know Sam is the selected contact that the app knows about.

To get this to work, you'll need to query the contact store and access the appropriate value in the `Cursor`.

So, let's get started!



Since Scott selected "Sam" from the contact list, getDisplayName should return Sam from the retrieved contact as positive reinforcement.

Query the contacts

Think of the contact store like a database. In fact think of the device having a big database with ***all*** of the content you can access on the phone and the contacts are inside there. You need a way to query that database though, and that's done with the `ContentResolver`.

You can retrieve the `ContentResolver` from your `Activity` using the `Activity` `getContentResolver` method.

```
ContentResolver contentResolver =
      getContentResolver();
```

Using the getContentResolver method to retrieve the default ContentResolver.

Then you can query the content provider passing in the `Uri` returned from the Contact selection screen.

The query returns a cursor just like a database query.

Query the ContentResolver.

```
Cursor cur = getContentResolver().query(
      intent.getData,
      null, null, null, null
);
```

Pass in the data from the intent.

The `ContentResolver` query return a `Cusor`, just like the cursor returned when you query a database.

Now let's see what content is in the Cursor.

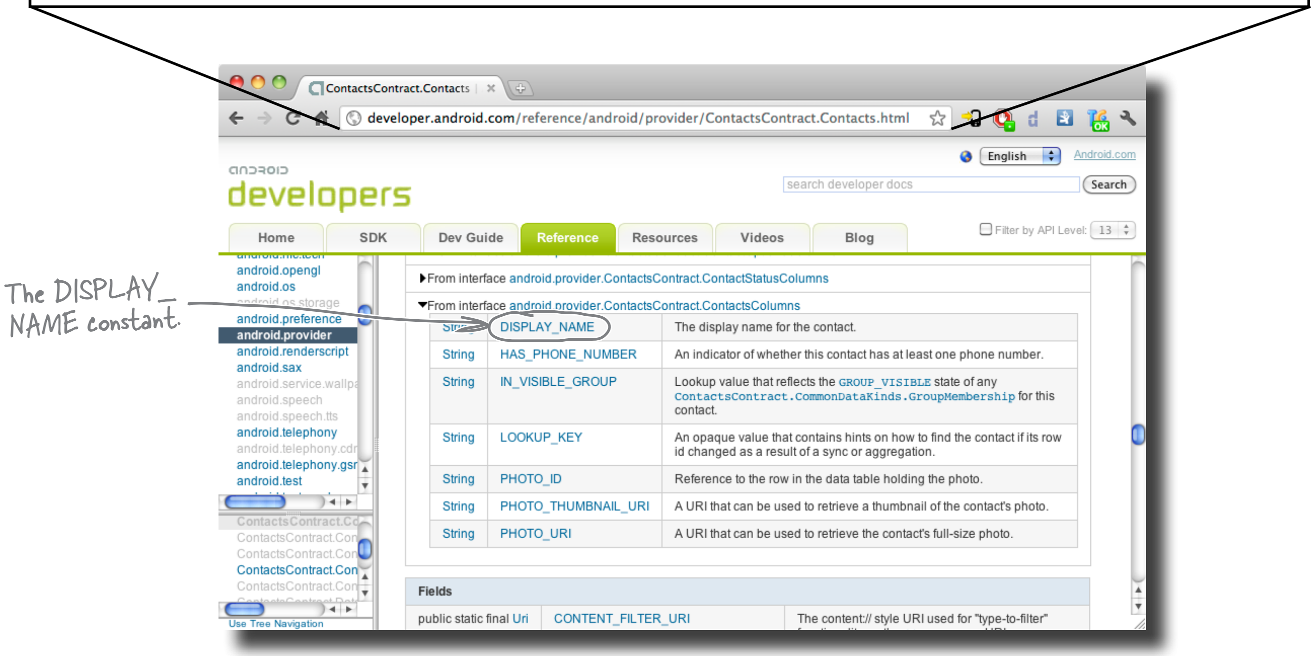
Cursor contents

Just like the Cursor database queries return, this Cursor is made up of a number of rows and columns. No columns were specified in this query so all the columns came back. This is resource intensive and you'll want to fix this. But for now, let's get the iteration working and then once you know the columns you need, you can query just for those.

How do you figure out what columns are coming back?

There are a few ways you could figure this out- you could write some code to print out the data or use the debugger. But before you do any of that, take a look at the documentation for `ContactsContract.Contact`. This class has a number of constants for the columns returned from the query, including one for `DISPLAY_NAME` which is what you'll need to display in the contact name field.

`http://developer.android.com/reference/android/provider/ContactsContract.Contacts.html`



Sharpen your pencil

Below is the updated `renderContact` method being passed in a Uri. If the `Uri` is not null, write the code to retrieve and set the display name on screen. To do this, you'll need to query the `ContentResolver` using the `Uri` passed in to `renderContact`. Then iterate through the cursor and retrieve the display name using constants. Remember, the `ContactsContract.Contact.DISPLAY_NAME` is a `String`. So retrieve the column index using the constant and retrieve the value. Also remember to use safe `Cursor` iteration and to close the `Cursor` when you're done.

```
private String getDisplayName(Uri uri) {
      String displayName = null;







      return displayName;
}
```

Put your code in here to query the contacts using the Uri, iterate the cursor, and set the value for the display name on the screen.

Sharpen your pencil Solution

Below is the updated `renderContact` method being passed in a `Uri`. If the `Uri` is not null, write the code to retrieve and set the display name on screen. To do this, you'll need to query the `ContentResolver` using the `Uri` passed in to `renderContact`. Then iterate through the cursor and retrieve the display name using constants. Remember, the `ContactsContract.Contact.DISPLAY_NAME` is a `String`. So retrieve the column index using the constant and retrieve the value. Also remember to use safe `Cursor` iteration and to close the `Cursor` when you're done.

```
private String getDisplayName(Uri uri) {
        String displayName = null;

        Cursor cursor = getContentResolver().query(uri, null, null, null, null);
                if (cursor.moveToFirst()) {
                        displayName = cursor.getString(
                                contactCursor.getColumnIndex(
                                        ContactsContract.Contacts.DISPLAY_NAME
                                )
                        );
                }
        cursor.close();

        return displayName;
}
```

Query the contacts with the uri passed in. Use getContentResolver to retrieve a ContentResolver.

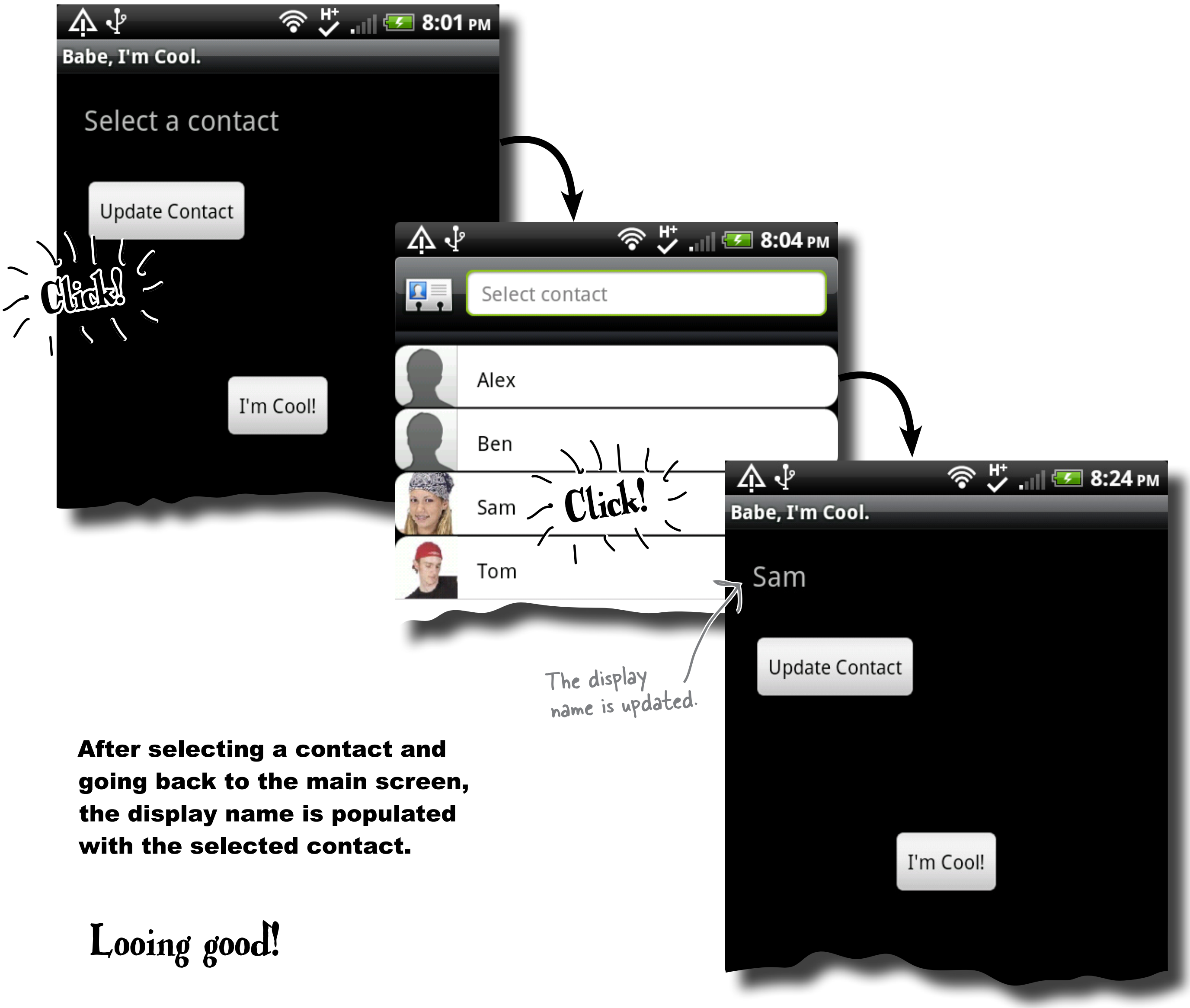
Move to the first row of the cursor (there should only be one)

Get the string value from the cursor, but first get the column index using the display name constant.



Test Drive

Run the app now and select a contact. The display name should be updated.



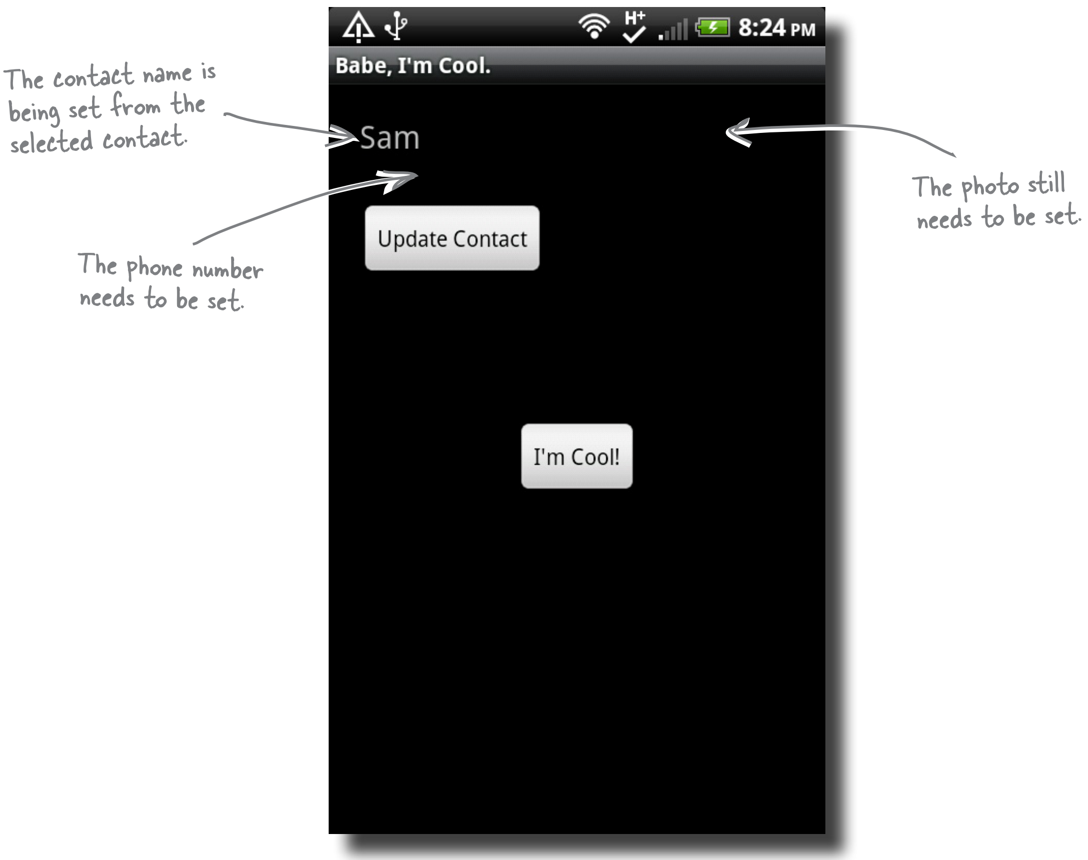
After selecting a contact and going back to the main screen, the display name is populated with the selected contact.

Looing good!

Display the phone number next

You've got the name displaying on the contact display on the main screen. This means you're successfully selecting a contact, getting the selected contact back and retrieving data values from that contact by querying the contact store. Whew!

Now you need to display the phone number and photo to complete the contact display.

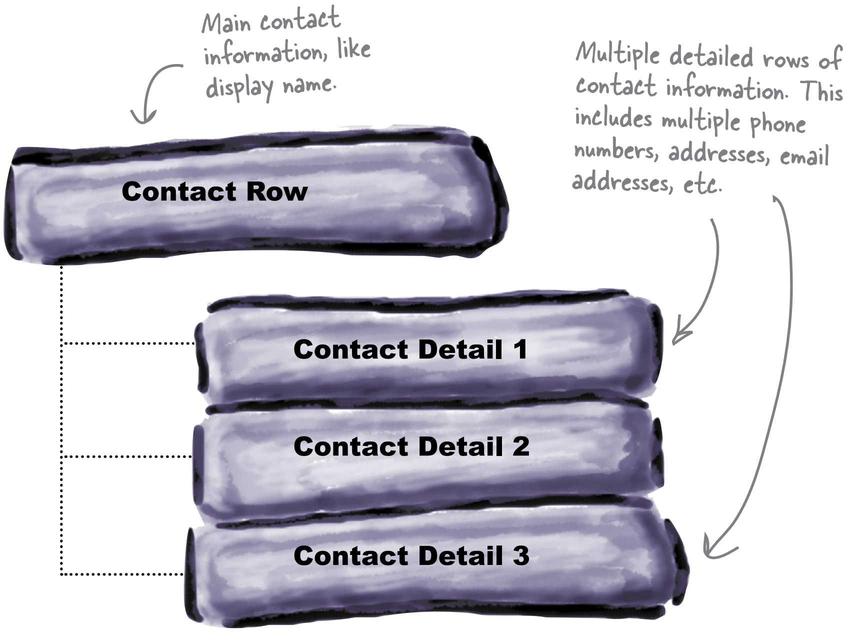


So what about these other fields?



Actually, retrieving the phone and picture are going to take a *little* more work...

Phone contacts can be a bit tricky. You can have multiple phone numbers (think home, mobile, office, etc), multiple addresses, etc. To handle this, contacts are actually implemented as separate rows. One row handles the main information for the contact (like the display name), and then there are multiple detail rows for the contact.



Turn the page to see how access the detail contact rows? →

Accessing contact info details

The general contact info row has some, but not all, of the information you need. This is pretty standard when you're working with the device contacts. The general row is about enough to make a list of contacts but that's it!

The content for the phone numbers in the contact detail table. It has a mix of all different kinds of numbers for multiple contacts.

Contact	Number	Type	Email
Ben	555-716-9333	Mobile	
Sam	555-299-2354	Work	
Alex	555-243-9786	Mobile	
Sam	555-867-5309	Mobile	
Sam	555-998-9125	Home	
Tom			awesometom@gmail.com

There are LOTS more columns at the end here ..

So how to get these detail rows?

The detail rows are also stored in the contact store and you can access them using another query to the `ContentProvider`. The `ContactsContract.CommonDataKinds` class contains a number of constants for working with these detailed rows. One in particular, `ContactsContract.CommonDataKinds.Phone.CONTENT_URI`, allows you to query just the phone numbers. All you need to do is pass in the Uri to the query method and you'll only get back phone numbers.

Implement getMobilePhone

Let's put this all in context and implement the `getMobilePhone` method. This method needs to query the contact details for the mobile phone associated with the selected contact. It will query the contact store using the Uri from `ContactsContract.CommonDataKinds` referencing the phone content.

Here is the method.

```
private String getMobilePhone(Uri uri) {
        String phoneNumber = null;
        Cursor phoneCursor = getContentResolver().query(
                ContactsContract.CommonDataKinds.Phone.CONTENT_URI,
                new String[] { ContactsContract.CommonDataKinds.Phone.NUMBER },
                null,
                null,
                null
        );
        if (phoneCursor.moveToFirst()) {
                phoneNumber = phoneCursor.getString(phoneCursor.getColumnIndex(
                        ContactsContract.CommonDataKinds.Phone.NUMBER)
                );
        }
        phoneCursor.close();

        return phoneNumber;
}
```

Pass in the content Uri constant for phone numbers.

Set the projection to the phone number.

Store the first phone number.

Close the cursor.

Return the phone number.



ImCool.java

BRAIN POWER

Something really important is missing from this method. Can you spot it? *(Hint: Look closely at the Uri passed in to getMobilePhone)*

Be selective with your contact query

If you used the getMobilePhone method as is in your app, you'll most likely get a phone number associated with a different contact than the contact selected. That doesn't make for a very good method!

The reason for this is that the ContactsContract.CommonDataKinds.Phone.CONTENT_URI used in the query refers to all phone records and you need to specify the contact you want.

You can refine results by adding a select value

There are additional parameters you can add to the query call that refine the results you'll get back. One of these is a String selection parameter. It acts like a **SQL WHERE** clause in the underlying query to the contacts. And just like a SQL WHERE clause, you can include **?**'s in the select String. Using another constant from the ContactsContract, your select parameter will look like this.

Another constant from the ContactsContract class that allows you to select a specific contact..

```
ContactsContract.CommonDataKinds.Phone.CONTACT_ID + " = ?"
```

This is going to be the ID for the contact.

With this select parameter, you also need to pass in an array of selection argument. These selection arguments will replace the **?**'s in the select string when the query is executed.

This is the ID for the contact you want to select.

```
new String[] { id }
```

The only issue now is that you don't have a reference to the contact ID in the getMobilePhone method. **But don't worry, you can query that too!**

Select just the numbers for your contact

Let's update getMobilePhone now. It needs first query the contact store to retrieve the ID of the selected contact based on the selected contact Uri. Then use that ID and pass it in through the selection arguments in the second query.

Passing in a projection of the contact ID. This will return only contact IDs

This first query retrieves the main contact row, and from that row you can retreive the ID.

Retrieve the selected contact's ID.

```
private String getMobilePhone(Uri uri) {
    String phoneNumber = null;

    Cursor contactCursor = getContentResolver().query(
        uri, new String[]{ContactsContract.Contacts._ID},
        null, null, null);
    String id = null;
    if (contactCursor.moveToFirst()) {
        id = contactCursor.getString(
            contactCursor.getColumnIndex(ContactsContract.Contacts._ID));
    }
    contactCursor.close();

    Cursor phoneCursor = getContentResolver().query(
        ContactsContract.CommonDataKinds.Phone.CONTENT_URI,
        new String[] { ContactsContract.CommonDataKinds.Phone.NUMBER },
        ContactsContract.CommonDataKinds.Phone.CONTACT_ID + " = ? ",
        new String[] { id },
        null
    );
    if (phoneCursor.moveToFirst()) {
        phoneNumber = phoneCursor.getString(phoneCursor.getColumnIndex(
            ContactsContract.CommonDataKinds.Phone.NUMBER)
        );
    }
    phoneCursor.close();

    return phoneNumber;
}
```

The second query retrieves the contact detail rows with the phone numbers of the selected onctact.

Pass in the select statement and selection arguments to select only phone numbers for the selected contact.

Just a little more refining to do

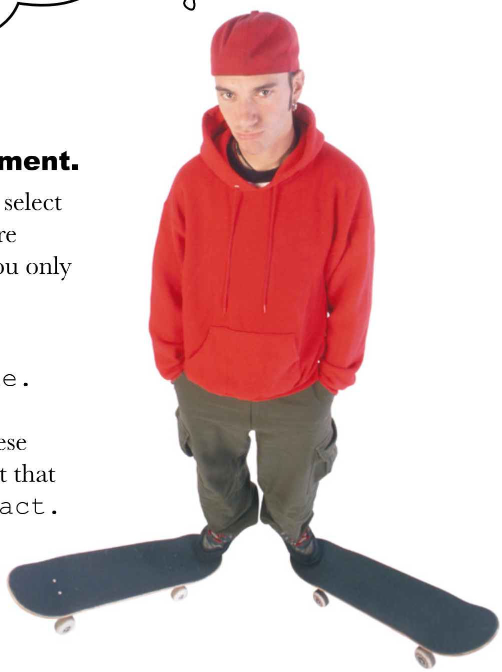
With the update of `getMobilePhone` to use a select statement using the selected contact ID, you'll only retrieve phone numbers for the selected contact. **This is good, but not good enough for you, a Head First rockstar!**

Here's the catch. This current `getMobilePhone` implementation retrieves all phone numbers for the selected contact. But for this app, you only want mobile phone numbers! You can't send a text message to a land line after all, so let's make sure we retrieve just the mobile numbers.

Now just Sam's detail rows are coming back, but we want to narrow that down even more to just mobile phone numbers.

Contact	Number	Type	Email
Sam	555-299-2354	Work	
Sam	555-867-5309	Mobile	
Sam	555-998-9125	Home	

So the phone numbers are narrowed down to just the selected contact. How do you make sure they are just mobile numbers now?



Get more specific with your select statement.

You're already selecting phone numbers by passing in a select statement to the query. Now you need to get a little more specific and add a clause to that select statement that you only want to select mobile phone numbers.

Luckily, there is a column referenced by the constant at `ContactsContract.CommonDataKinds.Phone.TYPE` that refers to the type of the phone number like mobile, home, or office. There are also constants for these different types in `ContactsContract`. The constant that refers to the mobile number type is `ContactsContract.CommonDataKinds.Phone.TYPE_MOBILE`.

Sharpen your pencil

Below is the query to retrieve the phone numbers for the selected contact. The second query from `getMobilePhone`. Update the code below adding a second clause to the select statement for the phone type to be mobile. Use `AND` to join the clauses in the select statement. Use the constants from the `ContactsContract`.

```
Cursor phoneCursor = getContentResolver().query(
        ContactsContract.CommonDataKinds.Phone.CONTENT_URI,
        new String[] { ContactsContract.CommonDataKinds.Phone.NUMBER },

        ContactsContract.CommonDataKinds.Phone.CONTACT_ID + " = ? ",

         ..............................................................

         ..............................................................

        new String[] { id },
        null
);
```

Add to this select statement to narrow the results down to only mobile phones.

Sharpen your pencil Solution

Below is the query to retrieve the phone numbers for the selected contact. The second query from `getMobilePhone`. You should have updated the code below adding a second clause to the select statement for the phone type to be mobile. You should have used `AND` to join the clauses in the select statement as well as using the constants from the `ContactsContract`.

Extend the select with AND.

```
Cursor phoneCursor = getContentResolver().query(
            ContactsContract.CommonDataKinds.Phone.CONTENT_URI,
            new String[] { ContactsContract.CommonDataKinds.Phone.NUMBER },

            ContactsContract.CommonDataKinds.Phone.CONTACT_ID + " = ? AND "
            + ContactsContract.CommonDataKinds.Phone.TYPE + " = "
            + ContactsContract.CommonDataKinds.Phone.TYPE_MOBILE,

            new String[] { id },
            null
);
```

Add the type constant.

Add = for the compare.

And finally, add the mobile type constant for comparison.



Test Drive

Run the app and select the contact again, and you should see the display name and the phone number display for your selected contact.



Now the display name AND the phone number are being displayed.

You've got two of the three renderContact helper methods working. You're almost there! Now it's just that photo...

Ready Bake Code

Now show the photo

There's a great helper method for loading the photo for a contact in `ContacsContact.Contacts` called `openContactPhotoStream`. You'll need to pass in the `ContentResolver` and a `Uri`. Notice that this Uri is using the `ContentUris.withAppendedId`. This is slightly different from the other `Uri` you've been using as it's actually generating a new `Uri` based on a stored constant plus the ID you're passing in Check out the online documentation for details.

```
private Bitmap getPhoto(Uri uri) {
        Bitmap photo = null;
        String id = null;
        Cursor contactCursor = getContentResolver().query(
                uri, new String[]{ContactsContract.Contacts._ID}, null, null, null);
        if (contactCursor.moveToFirst()) {
                id = contactCursor.getString(
                        contactCursor.getColumnIndex(ContactsContract.Contacts._ID));
        }
        contactCursor.close();
        try {
                InputStream input =
                        ContactsContract.Contacts.openContactPhotoInputStream(
                                getContentResolver(),
                                ContentUris.withAppendedId(
                                        ContactsContract.Contacts.CONTENT_URI,
                                        new Long(id).longValue())
                        );
                if (input != null) {
                        photo = BitmapFactory.decodeStream(input);
                }
                input.close();
        } catch (IOException iox) { /* exception handing here */ }
        return photo;
}
```

A similar query to getMobilePhone to retrieve the ID. (points to the block from `String id = null;` to the closing `}` of the `if`)

Create an InputStream using the helper method. (points to `ContactsContract.Contacts.openContactPhotoInputStream(`)

Use BitmapFactory to decode the stream into a real, live bitmap! (points to `photo = BitmapFactory.decodeStream(input);`)

Return the photo bitmap. (points to `return photo;`)

Test Drive

Run the app and select a contact one last time. You should see all three fields update- the display name, the phone number AND the photo.



Great work! All three methods are working to retrieve and display the contact info!

Getting ready to send the text message

The last feature to build before you can give the app to your users for testing is to send the text message. Pressing the "I'm Cool" button should trigger the text message, so before going any further, let's add an `onClick` attribute to the I'm Cool button on screen and invoke a method called `onImCoolButtonClick` in the Activity.

```
<Button android:id="@+id/im_Cool"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:text="I'm Cool"
        android:onClick="onImOkButtonClick"
        />
```

Add the onClick property to the I'm Cool button in main.xml.



main.xml

Also add the method to the activity that's called by the onClick attribute.

```
public void onImCoolButtonClick(View view) {

        The code to send the text
        message will go in here.

}
```



ImCool.java

How to send a text message

Sending a text message on Android couldn't be easier. There is a class called `SmsManager` with a method that sends a text message. As long as your app is configured with proper permissions to send text messages (using the `android.permission.SEND_SMS` permission) you can send text messages to whoever you like!

Take a look at the `sendTextMessage` method.

```
sendTextMessage (
        String phoneNumber,
        String serviceCenterAddress,
        String text,
        PendingIntent sentIntent,
        PendingIntent deliveryIntent
)
```

The phone number to send the text message to.

The message text.

These are special intents that can be activated like callbacks. You won't need to use them for basic text message sending.



Make sure to add the `SEND_SMS` permission.

If you don't add the SEND_SMS permission to your app and run it on a device, you'll get an error about missing permissions. Stop now and add the `android.permission.SEND_SMS` *permission to your AndroidManifest.xml file.*

Add the action method for the I'm Cool button

Let's make one small change to your Activity to send text messages. Right now, the `contactUri` is used to update the display after selection, but it's not stored anywhere. For now, store the `contactUri` in your `Activity` as an instance variable.

```
public class ImCool extends Activity {
       private static final int PICK_CONTACT_REQUEST = 0;
       private Uri contactUri;
```

← Add a variable for the contactUri.

Now store the `Uri` of the selected contact when it's passed back from the contact selection in `onActivityResult`. This way, you'll be able to call `getMobilePhone` to retrieve the selected contact's phone number to send the text message.

```
protected void onActivityResult(int requestCode, int resultCode, Intent intent) {
       if (requestCode == PICK_CONTACT_REQUEST) {
              if (resultCode == RESULT_OK) {
                     contactUri = intent.getData();
                     renderContact(contactUri);
              }
       }
}
```

← Cache the contactUri that comes back from the contact selection.



ImCool.java

Pool Puzzle



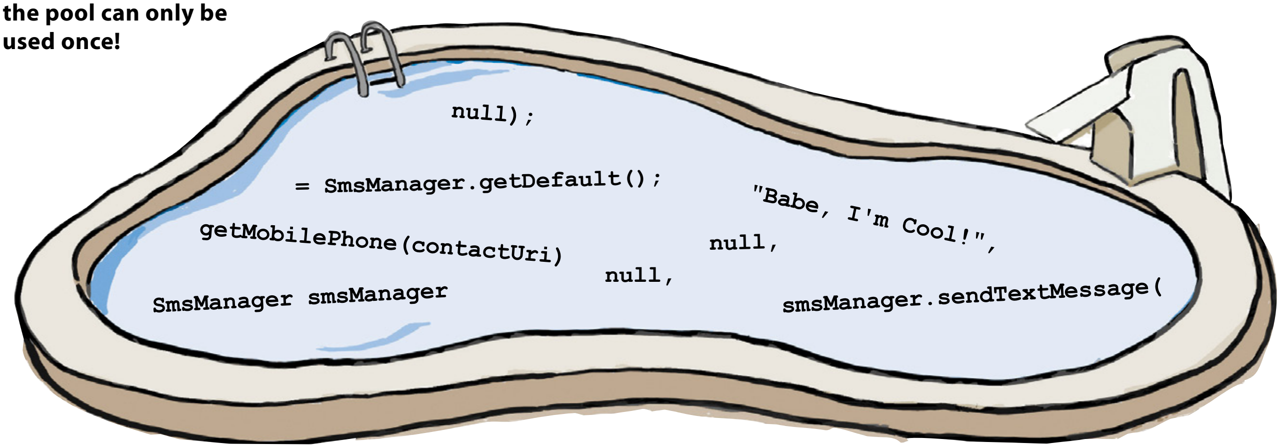
Your **job** is to take the code fragments from the pool and place them into the empty `onImCoolButtonClick` method. You may **not** use the same code fragment more than once, and you won't need to use all the code fragments. Your **goal** is to make the `onImCoolButtonClick` send the text message to the selected contact.

```
public void onImCoolButtonClick(View view) {




}
```

Note: each thing from the pool can only be used once!



Pool Puzzle Solution



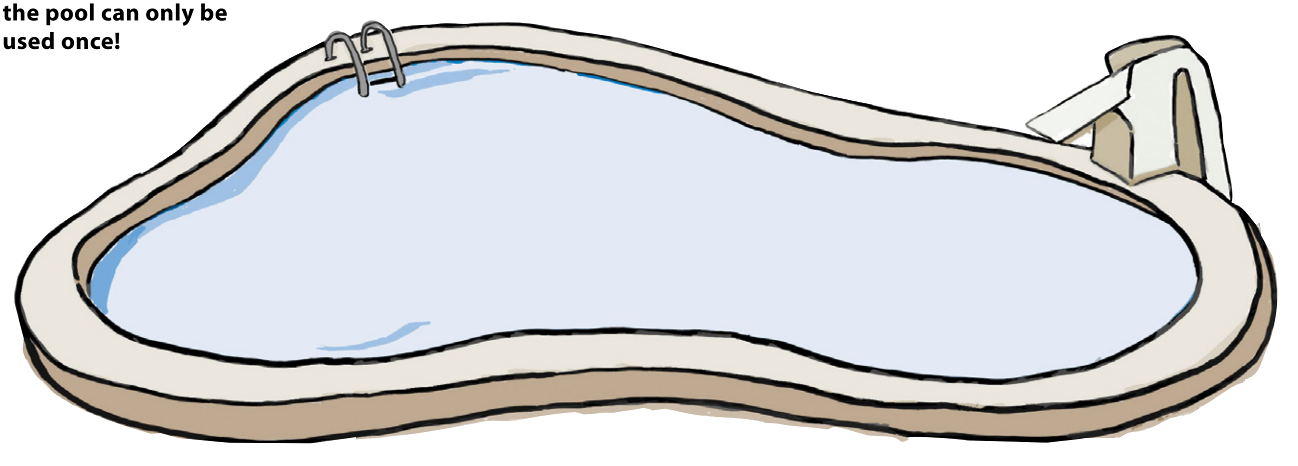
Your **job** is to take the code fragments from the pool and place them into the empty `onImCoolButtonClick` method. You may **not** use the same code fragment more than once, and you won't need to use all the code fragments. Your **goal** is to make the `onImCoolButtonClick` send the text message to the selected contact.

```
public void onImCoolButtonClick(View view) {

    SmsManager smsManager  = SmsManager.getDefault();

    smsManager.sendTextMessage(
            getMobilePhone(contactUri),
            null,
           "Babe, I'm Cool!",
           null,
           null);
}
```

Note: each thing from the pool can only be used once!





Go Off Piste

Now that you have the hang of `ContentProviders`, here are some other cool things to look into.

Audio Content

Using the same concepts you learned searching for contacts, you can load audio too! Check out the android.provider.MediaStore.Audio for more information on loading music, playlists, album covers, and more.

Photo and Video

Loading photos and videos from the device works in a similar way too. Check out MediaStore.Images and MediaStore.Video for more.

Modify Data

Content providers aren't just read only, you can modify content too. For example, writing or modifying a phone number, adding a new photo, and more. Check out the docs for more information.

Many, many, more...

Take a look at the android.provider package for even more content you can access from your apps.

Write your own!

If you have data in your app that you'd like to share with other apps, you can build your own content provider that other apps can query. Take a look at http://developer.android.com/guide/topics/providers/content-providers.html for more information.



Your Android Toolbox

The app is now functional! You implemented contact selection using native behavior and queries all of the contact details to render in the display. You also implemented text messaging and tested on a real world device.

Invoking native behavior

- Intents can invoke specific Activities, or a Uri/Action combination. If you pas in this Uri/Action commbo, the OS determines the most appropriate Activity to respond to the Intent.
- Use constants from Intent for these Uris and Actions
- Use startActivity or startActivityForResult as needed for your app.

Querying Contacts

- Use ContentResolver and constants from ContactsContract to query contacts.
- Query general contact information using the full contact Uri returned form selecting a contact
- Query contact details with the help of contact constants in ContactsConstract subclasses
- Refine your queries with select statements to get the data you want

BULLET POINTS

- Use native behavior by invoking an `Intent` with `Actions` and `Uris` instead of explicit `Activity` references.
- Check the online documentation for `Intent` to see which `Uris` have native responders.
- Use constants for `Actions` and `Uris` whenever possible. This way you'll be prepared when things change.
- Make sure to add the appropriate permissions for your app, this one needed `READ_CONTACTS` and `SEND_SMS`.
- Sometimes it's easier to test on the emulator and sometimes it's easier to test on hardware. Do what makes sense for you app. And make sure not to only test on the emulator since you're deploying your app to the real world, NOT the emulator!
- Contact information is located in an on-device data store you can query like a database.
- Contact queries return `Cursor`s, just like a database query.
- Contact information is stored in spearate records for main contact information and contact details.
- Query contact (and other OS stored information) information using `ContentResolver`.
- Easily send text messages from your app using `SmsManager` (and adding the `SEND_SMS` permission)

12 advanced graphics

Make your apps shine



With all the competition in the marketplace, your apps need to do more than just work... they need to look great doing it! For some of your more basic apps, using the stock Android look and feel is fine. But when you want to built great looking apps that really **wow your users and customers**, you're going to need to need to use ***graphics***. In this chapter, you'll learn two *advanced techniques for adding images to your apps*. First you'll learn how to use images on your buttons. Then you'll learn how to use special resizable images that will really help your apps look ***fabulous*** on all kinds of different screen sizes.

It needs to be even better

Sam dropped in while you were finishing up the message sending and asked for a quick look at the app. After showing it to her, it became clear that function alone is not enough. It needs to look great too.



Good apps need good graphics

You might be a strong engineer and a great graphics designer. And if you can design and build your own apps, this is where you'd open up your favorite graphics tools and create some great graphics to make the app look super slick. But if you're like the rest of us, you're going to need some help.

Don't worry though, with the super high quality graphics in even the most basic apps, getting outside graphics help is pretty standard these days!

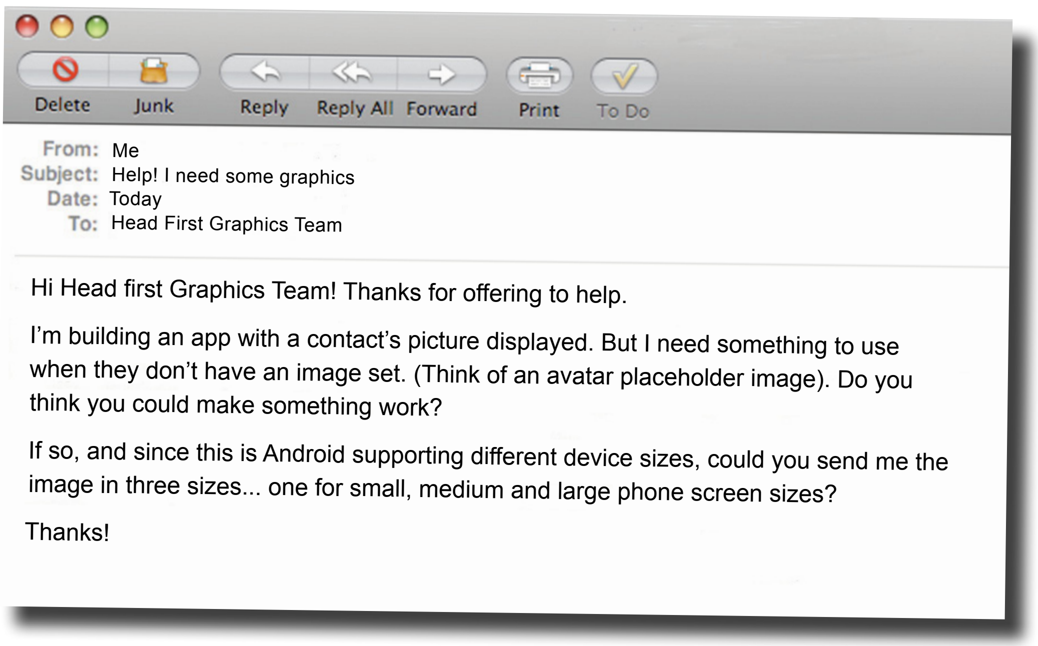
Let's see if there is anyone who can help us out with this...

Meet the Head First Graphics Team

Turns out there's a great group of graphics artists *just dying* to **help you on your latest project**! They just need you to ***send them an email*** describing what you need.



Here's your email to the Head First Graphics Team



Delete Junk Reply Reply All Forward Print To Do

From: Me
Subject: Help! I need some graphics
Date: Today
To: Head First Graphics Team

Hi Head first Graphics Team! Thanks for offering to help.

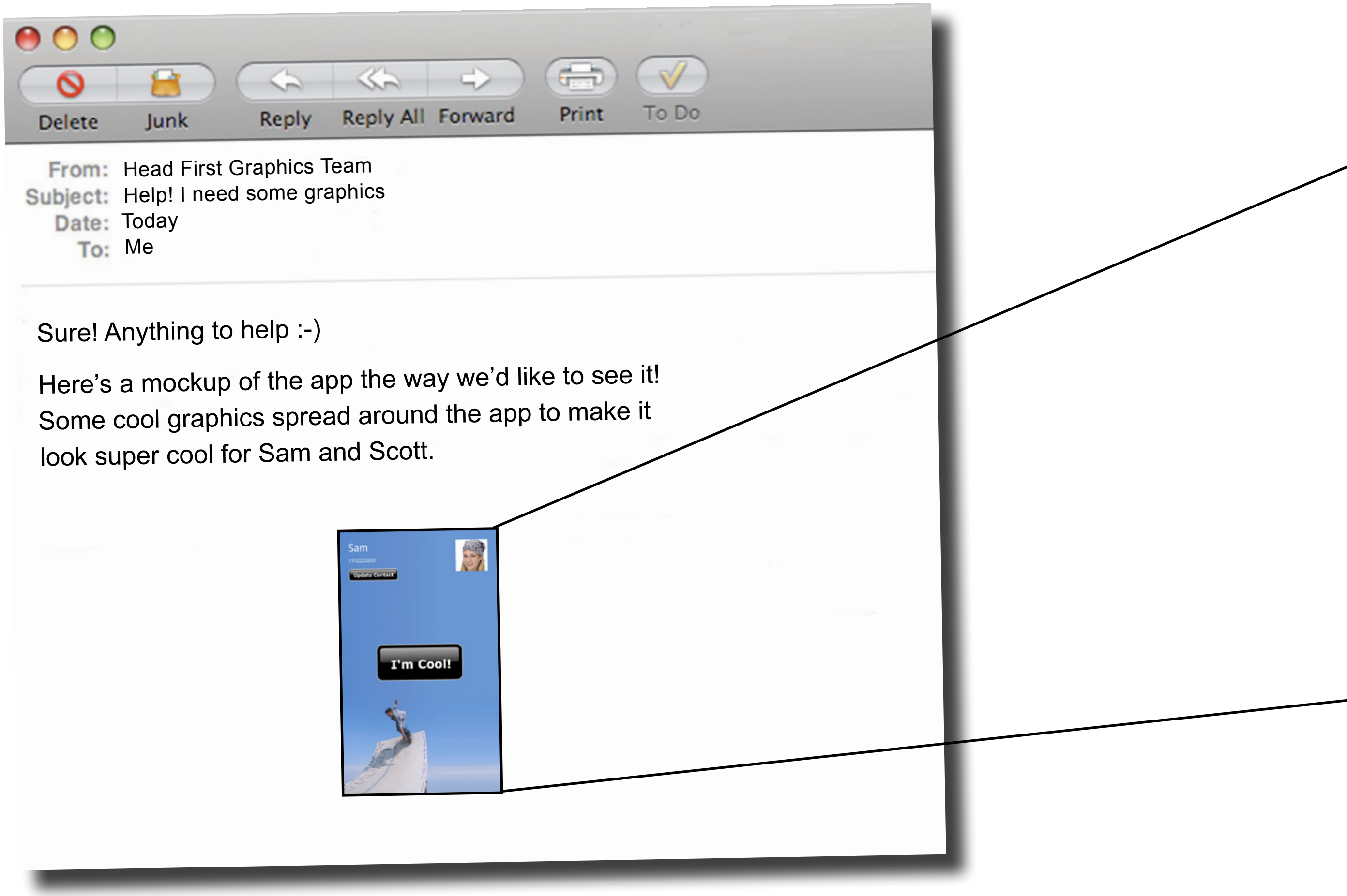
I'm building an app with a contact's picture displayed. But I need something to use when they don't have an image set. (Think of an avatar placeholder image). Do you think you could make something work?

If so, and since this is Android supporting different device sizes, could you send me the image in three sizes... one for small, medium and large phone screen sizes?

Thanks!

Give the app some polish

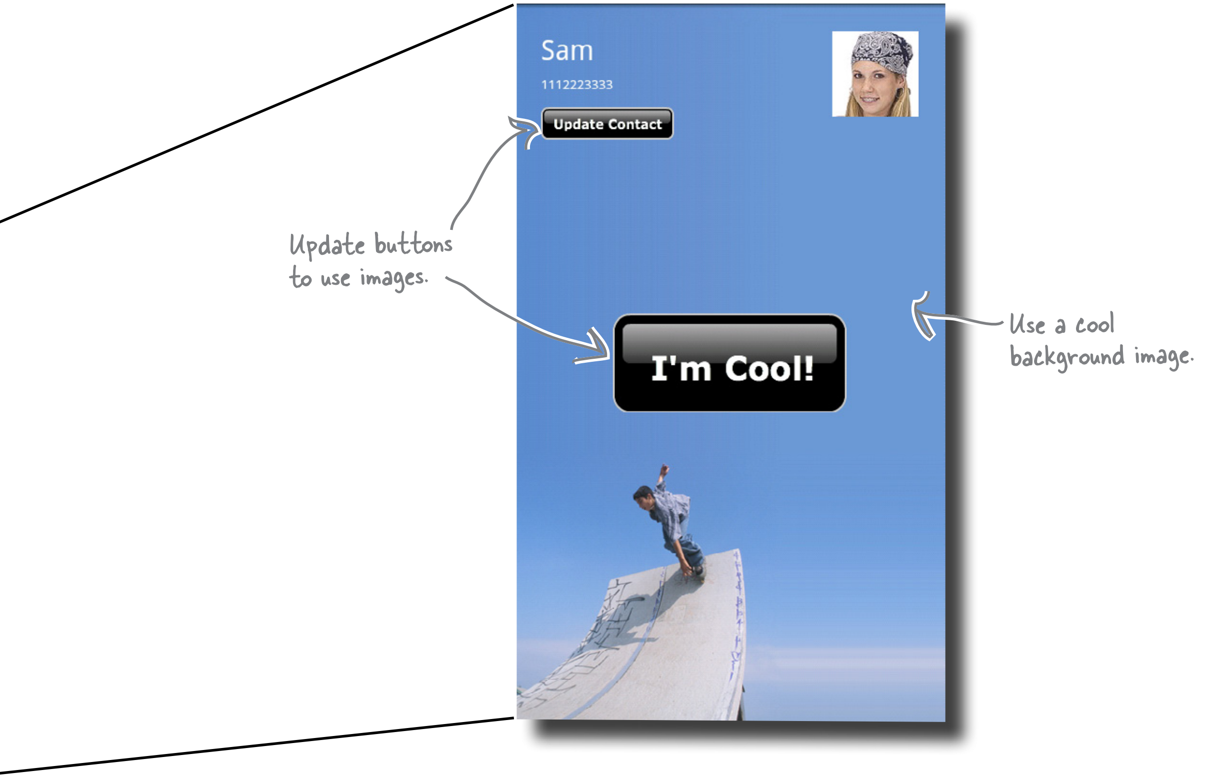
The Head First Graphics Team just got back to you with a sketch of how to update the design of the app. Let's take a look at their design and see what it would take to implement it.



Now let's see what needs to be done to make your app look like this picture.

Use button images...

This design uses custom images for both the **Update Contact** and **I'm Cool** buttons. You'll need to update the current buttons to ***use images***, and you'll need to get those image resources (the actual images for the buttons) from the Head First Graphics Team.



...and use a background image

This design has a background image that stretches across the entire screen. You'll need to get this **image** from the graphics team and set it on the ***background***. The issue here is that you ***don't really know the actual size of the screen***. Even if you know the screen grouping, the actual screen might be a a few more or less pixels than you're expecting. To solve this, you'll need to use a *special kind of image* that can **resize**.

Turn the page to get started →

Use image buttons instead of plain, boring ones

Let's start implementing the Head First Design Team's graphical update by adding the images for the buttons.

Android provides a special button `View` called `ImageButton` specifically for buttons with images. To use `ImageButton`, just declare a `View` of type `ImageButton` and instead of setting the text, set the `android:src` attribute to reference an image.

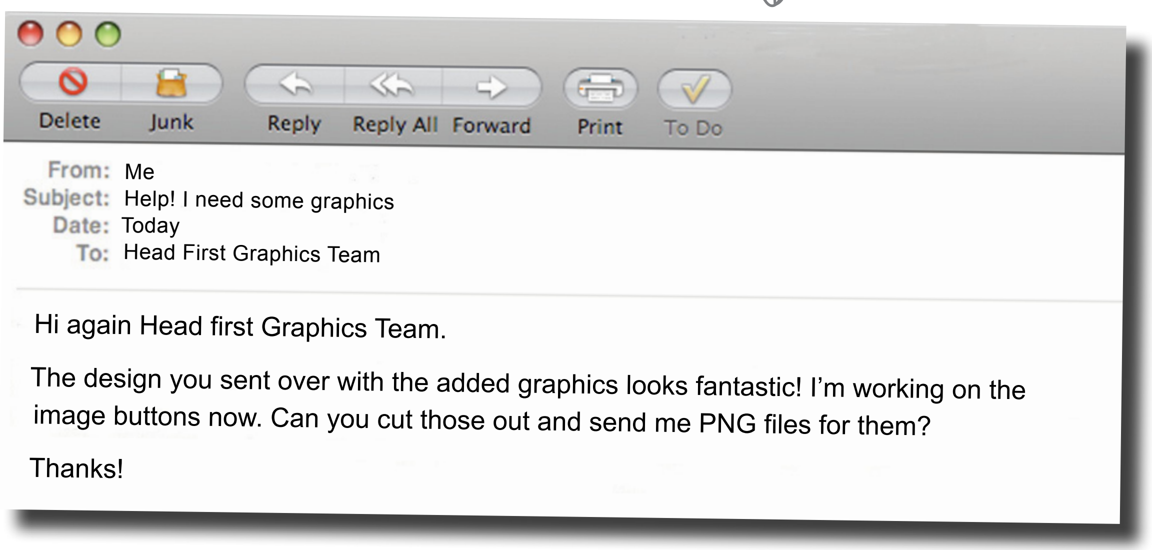
The android:src attribute should reference an image

The view type is ImageButton

```
<ImageButton
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/my_image_name"
        />
```

Before you can add the ImageButtons to your layout, you need the images to use. Time for another email to the Head First Graphics Team asking for images from their mockup.

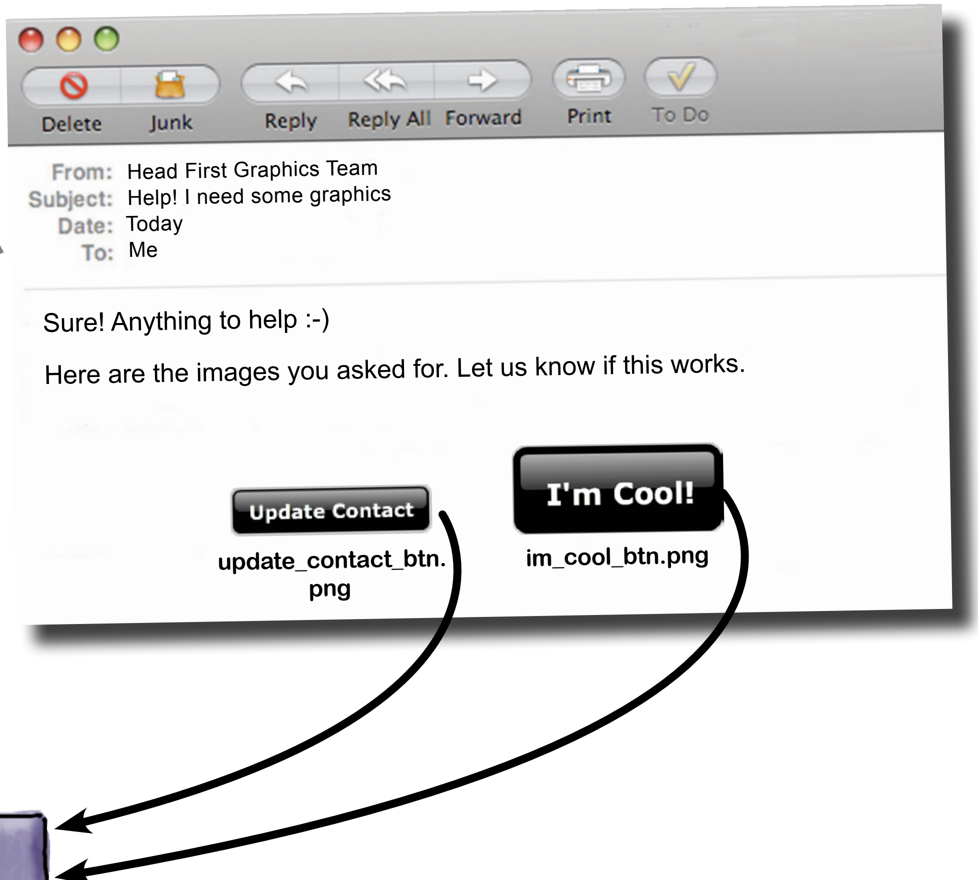
Your latest email to the graphics team.

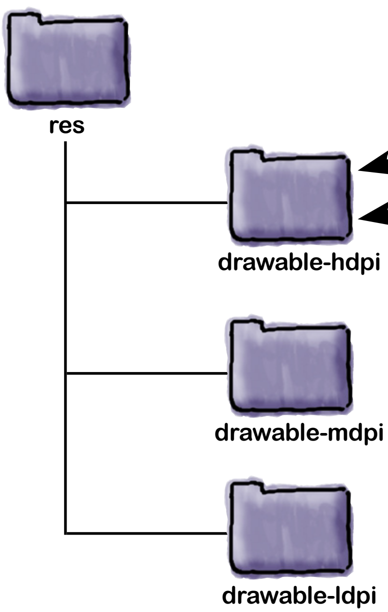


Add the images to your project

The Head First Graphics Team got back to you and sent you two images. Add them to your project under the res directories in `drawable-hdpi`.

Here's your reply from the Graphics team!





Watch it!

Make sure to cover ALL of the screen resolutions you're targeting.

This chapter is just targeting high resolution screens. As you're building your own apps, you'll need to add the button images for each resolution you're targeting. You can still use just one selector file though.

Add the ImageButton

With the new button images added to the res directory, update `main.xml` replacing the regular `Buttons` with `ImageButtons`. Set the `android:src` attribute to the names of the images you added from the graphics team. Also, remove the android:text attributes from both buttons since the images both have styled text embedded in them.

Change the view type to ImageButton.

```
<ImageButton android:id="@+id/update_contact"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Update Contact"
      android:layout_below="@+id/contact_phone"
      android:layout_alignLeft="@+id/contact_name"
      android:layout_marginTop="10dp"
      android:src="@drawable/update_contact_btn_bkg"
      />
```

remove the android:text attribute.

Change the view type to ImageButton.

```
<ImageButton android:id="@+id/im_ok"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_centerInParent="true"
      android:text="I'm Cool!"
      android:src="@drawable/im_cool_btn_bkg"
      />
```

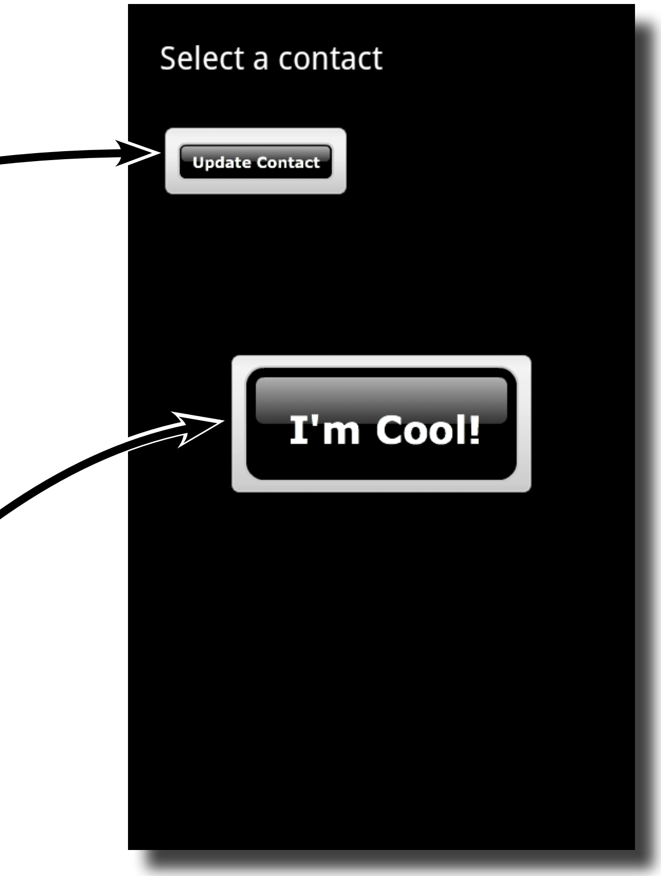
Set the android:src to the names of the images you added (without the .png extension)



main.xml

Test Drive

Now that you have the images added to your project and the `ImageButtons` added to your layout, run the app and see how it looks!







The images are displaying, but you've got some cleanup to do...

Remove the background

The image on an ImageButton doesn't cover the entire button. The ImageButton has a default background and the image you set in the android:src attribute is drawn on top of it. That's why you have that weird border. If you set the background to null, you'll just see you're image.

```
<ImageButton android:id="@+id/update_contact"
    android:background="null"
```

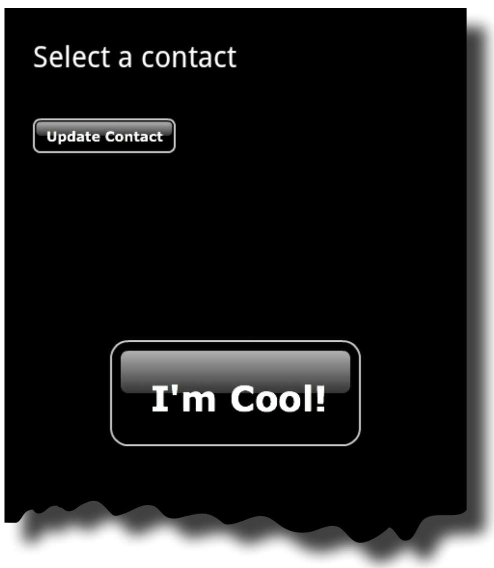
Set both android:background attribute values to null.

```
<ImageButton android:id="@+id/im_cool"
    android:background="null"
```

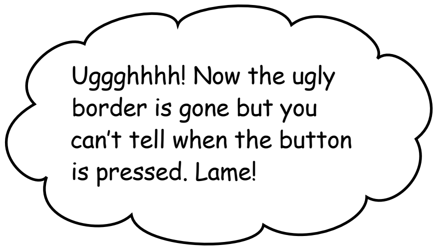


main.xml

Now take a look back at the app, and notice that the ImageButton borders are gone. All you can se now is the image drawable from the android:src attribute.



There's a lurking problem though. Try pressing one of the ImageButtons now...



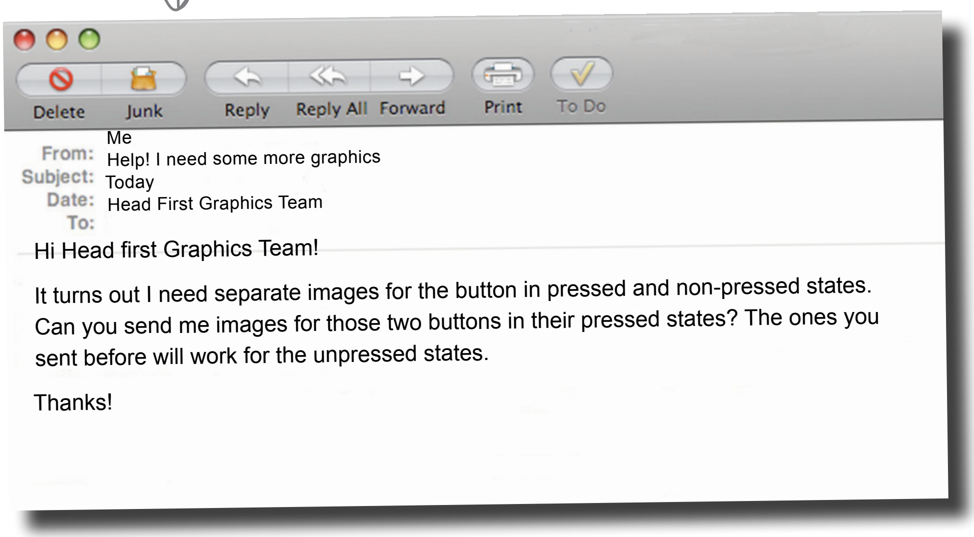


Lame, yes. But fixable!

By default, the button indicates it's being pressed by changing the background to orange. The indication that the button is pressed is really important for your users, but the big gray box around your great new images looks awful! **What to do?**

The solution is to have two *different* images: one for when the button is pressed and one for when it isn't. And since you need more images, that can only mean one thing... another email to the Head First Graphics Team!

Another email to the graphics team...


Delete Junk Reply Reply All Forward Print To Do

From: Me
Subject: Help! I need some more graphics
Date: Today
To: Head First Graphics Team

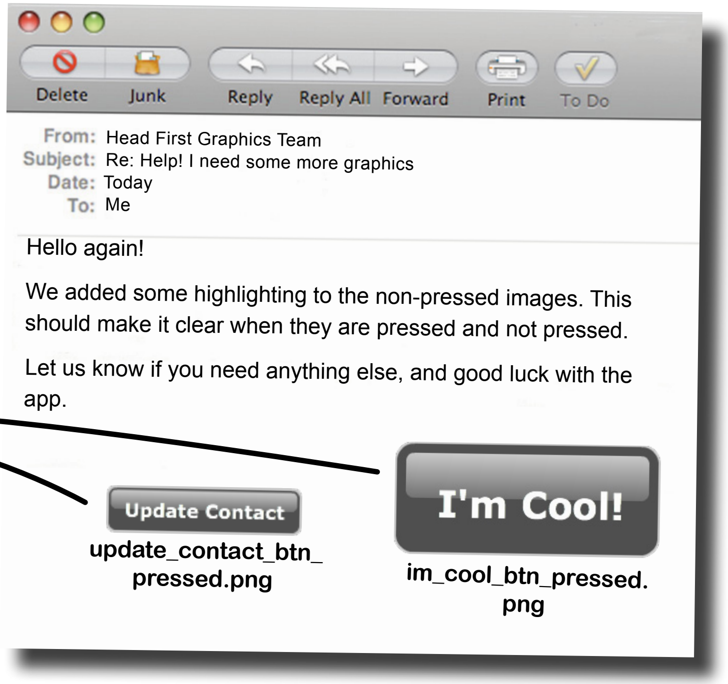
Hi Head first Graphics Team!

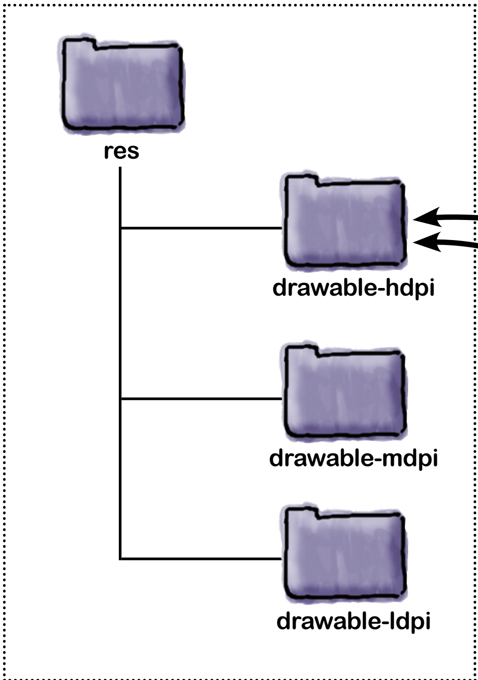
It turns out I need separate images for the button in pressed and non-pressed states. Can you send me images for those two buttons in their pressed states? The ones you sent before will work for the unpressed states.

Thanks!

Add the (new) images to your project

Looks like Head First Graphics Team just got back to you! Let's plug in the images they sent back.





Using different images for button states

There is only one attribute - `android:src` - to set the image on an `ImageButton`. But you want to use two ***different*** images: one when the button is pressed and another one when the button is in it's normal state. You could add a listened to the button and change the image displayed when pressed, but there is a much easier way!

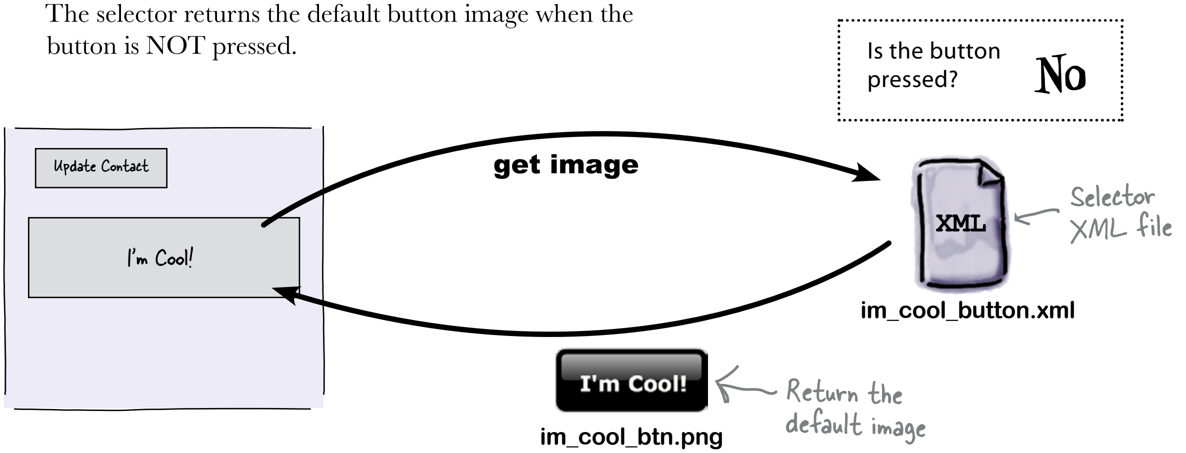


Use selectors to control button images

Selectors allow you to define **multiple** images to use for buttons **based on state**. Selecters are implemented as XML files with elements inside the file referring to specific states, and which image to use for that state. Then you can set the selector as the drawable instead of a specific image, and the `ImageButton` will automatically select and update the image according to its state.

1 **When the button is not pressed**

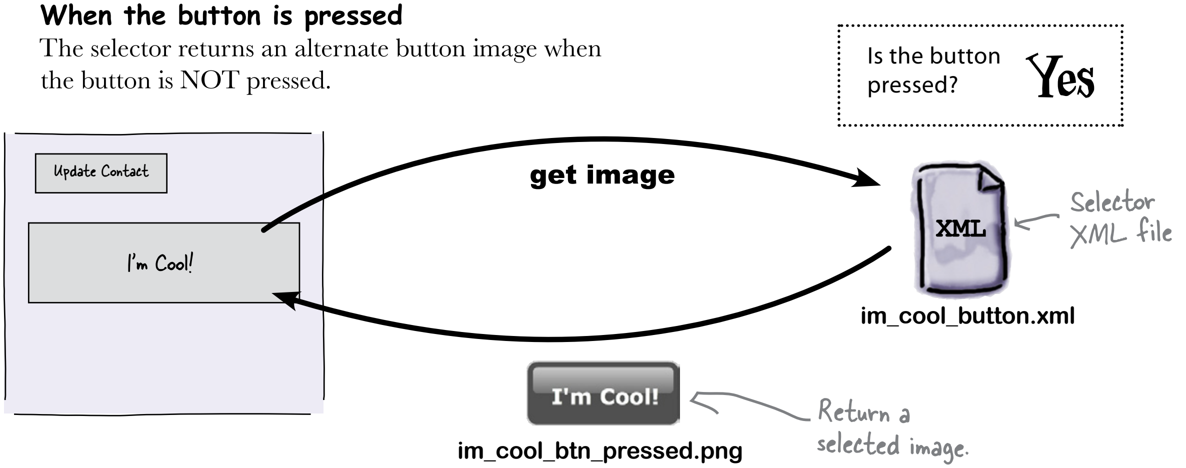
The selector returns the default button image when the button is NOT pressed.



---

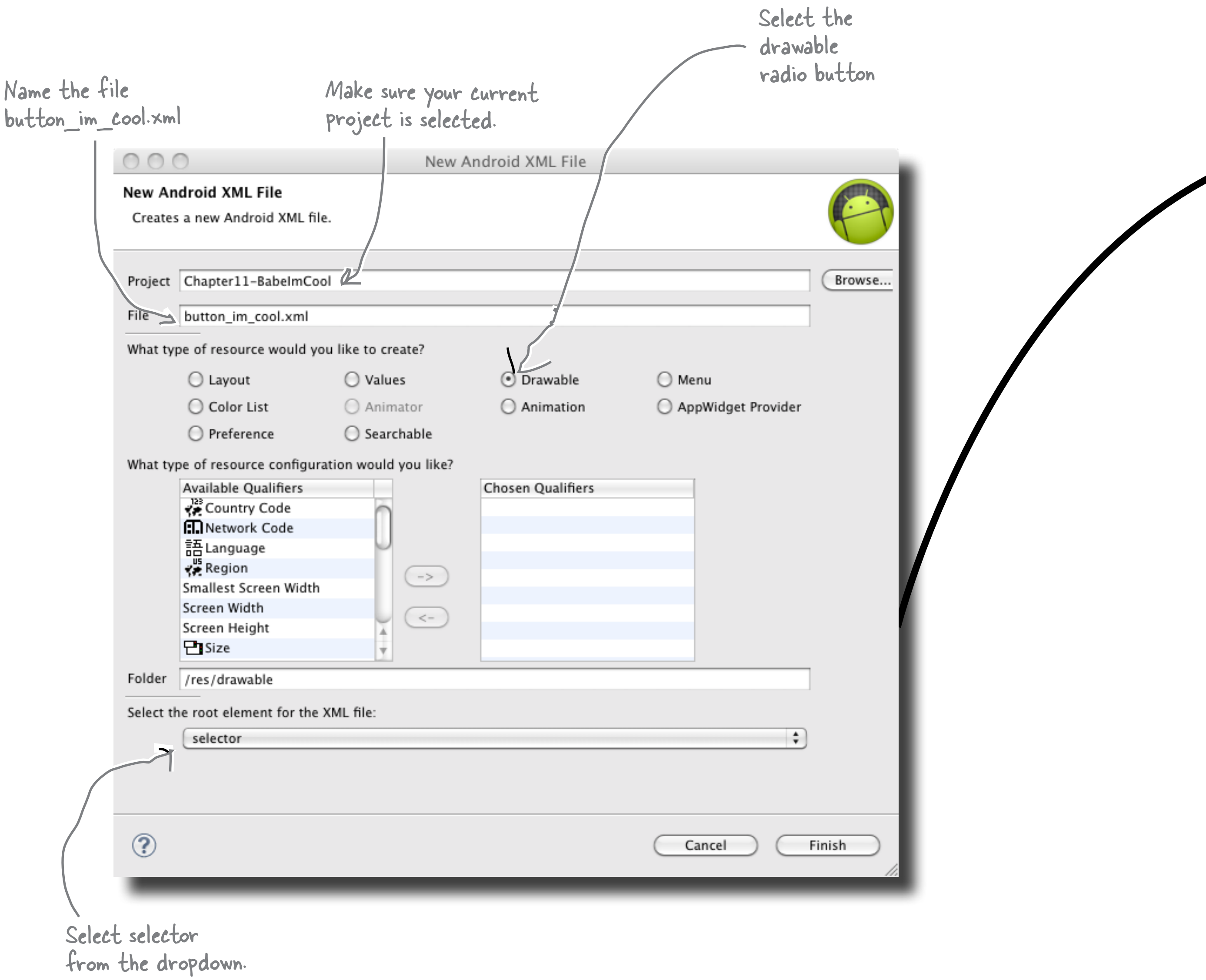
2 **When the button is pressed**

The selector returns an alternate button image when the button is NOT pressed.



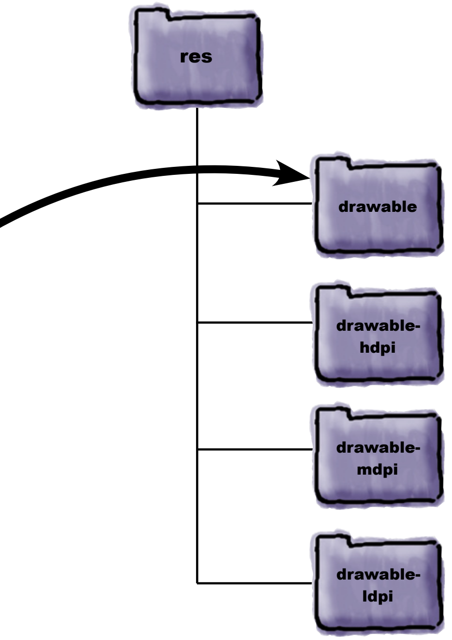
Make a new selector file

Start by making a new selector XML file. You can create one using the same wizard that you use to create new Android layouts and other XML files. Select **File → New → Android XML File** to launch the wizard.



Where is the selector file?

The new empty selector XML file was created by default in the `res/drawable` directory.



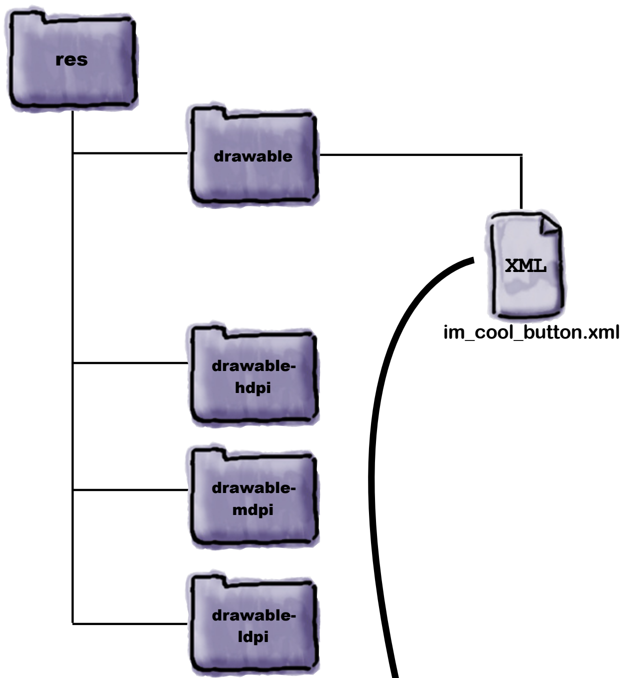


Selectors are just pointers to files, and the Android runtime finds the right ones.

By default, Android looks in the drawable directory fro the selector files. But when it comes to loading an image, the Android runtime tries to load images first from the resource folder specific to the screen size group. So in this case, the selector XML can live perfectly happy in the drawable folder and be found by the runtime. But when an image gets loaded, the runtime starts by looking in the drawable-hdpi folder (*assuming you're running on a high resolution device*) and loads the image from that folder.

Open the new selector file

Navigate to `res/im_cool_button.xml` and double click to open it. The autogenerated file starts out with an empty selector.



Do this!

Navigate to the `im_cool_button.xml` in the Package Explorer. You can fund it under the `res/drawable` directory. Double click top open it and you'll see an empty selector file.

Here is the (currently) empty selector declaration.

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">

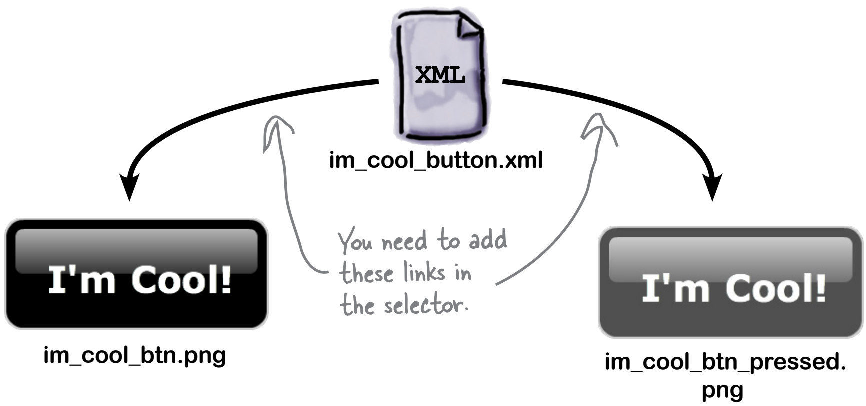
</selector>
```

You'll be completing the selector by adding images and states here.



Add the image pointers

The autogenerated selector is empty and you need to add the links from specific states to the images that should be displayed for them. You're going to need two image/state combinations: one for when the button is pressed and one when it's not.



Add items to the selector

Linking a state to a drawable inside selectors is done by defining `<item>` elements. Here is an `item` element that will render the `im_cool_button.png` image when the button is pressed.

This item is in effect when the button is pressed.

```
<item android:state_pressed="true"
    android:drawable="@drawable/im_cool_button" />
```

Display the im_cool_button drawable when the item is in effect.



Selector Magnets

Below is the empty selector from `button_im_cool.xml`. Use the code magnets to add two items to the selector. Add one item to show `im_cool_btn_bkg.png` when the button is not pressed. And add another item to show `im_cool_btn_bkg_pressed.png` when the button is pressed.

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">



</selector>
```

Add the two items here.



button_im_cool.xml

Your magnets.

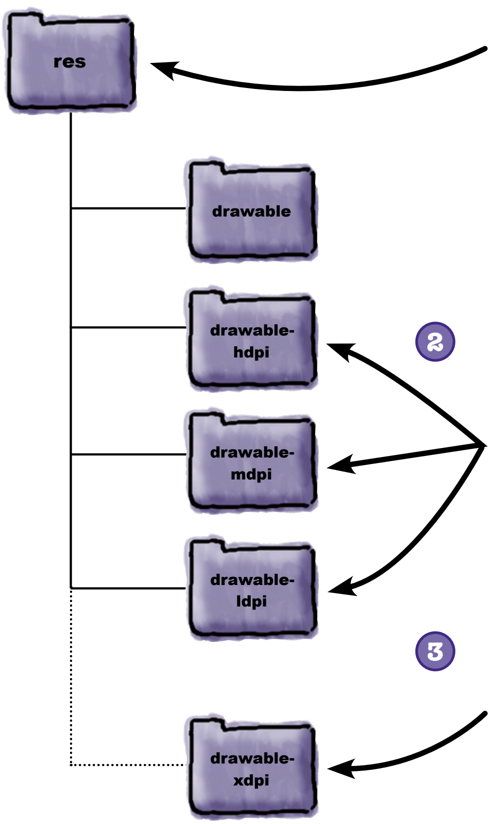
```
<item
android:state_pressed="true"
android:drawable="@drawable/im_cool_btn_bkg"
/>
android:state_pressed="false"
android:drawable="@drawable/im_cool_btn_bkg_pressed"
/>
<item
```



Geek Bits

This selector will only work on devices that fall in the HDPI category. This is fine (for now) since we know Sam and Scott both have HDPI Android phones. With that in mind, you should ***do careful analysis*** of your target users and **make sure to cover their devices as well**.

Here are some tips to cover as many devices as possible.



1. **Put default drawable images in drawable**
The other folders contain device group specific images. But if an image resource isn't found for the specific device group (or a new group is introduced) the Android runtime will check here.

2. **Add images for each screen group**
Each screen grouping will try and load its pecific images. Make images for every group and add them to appropriate folder.

3. **Look for new screen groups**
With the addition of tablets and larger phones, the number of screen groups is growing. Keep an eye out for these new groups and make sure you have the resources you need for these groups.

And remember, you don't need resources for *every single* resolution. You might find that with flexible layouts and decent scalable images, you can get away with really great **hdpi** and **mdpi** images and you're all set. Don't do more work then you have to, but do make sure your app looks great on all devices.



Selector Magnets Solution

Below is the empty selector from `button_im_cool.xml`. You should have used the code magnets to add two items to the selector. The first item should show `im_cool_btn_bkg.png` when the button is not pressed. And the other item to show `im_cool_btn_bkg_pressed.png` when the button is pressed.

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">
```

This item is in effect when the button is NOT pressed, and it displays the im_cool_btn_bkg drawable.

```
    <item android:state_pressed="false"
          android:drawable="@drawable/im_cool_btn_bkg" />
```

This item is in effect when the buttis is pressed, and it displays the im_cool_btn_bkg_pressed drawable.

```
    <item android:state_pressed="true"
          android:drawable="@drawable/im_cool_btn_bkg_pressed" />

</selector>
```



button_im_cool.xml

Set the selector as the button's drawable

The selector is a drawable, so you can set it as the background just like using an image. The last step before testing the selector is to set the `android:src` attribute on the `ImageButton` to the selector instead of pointing directly to an image drawable.

```
<ImageButton android:id="@+id/im_ok"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:src="@drawable/button_im_cool"
        android:background="@null"
        />

</RelativeLayout>
```

Set the selector as the drawable for the ImageButton src.

main.xml

there are no Dumb Questions

Q: These selectors look cool, but what if I want to use a different image when the button is, say, disabled?

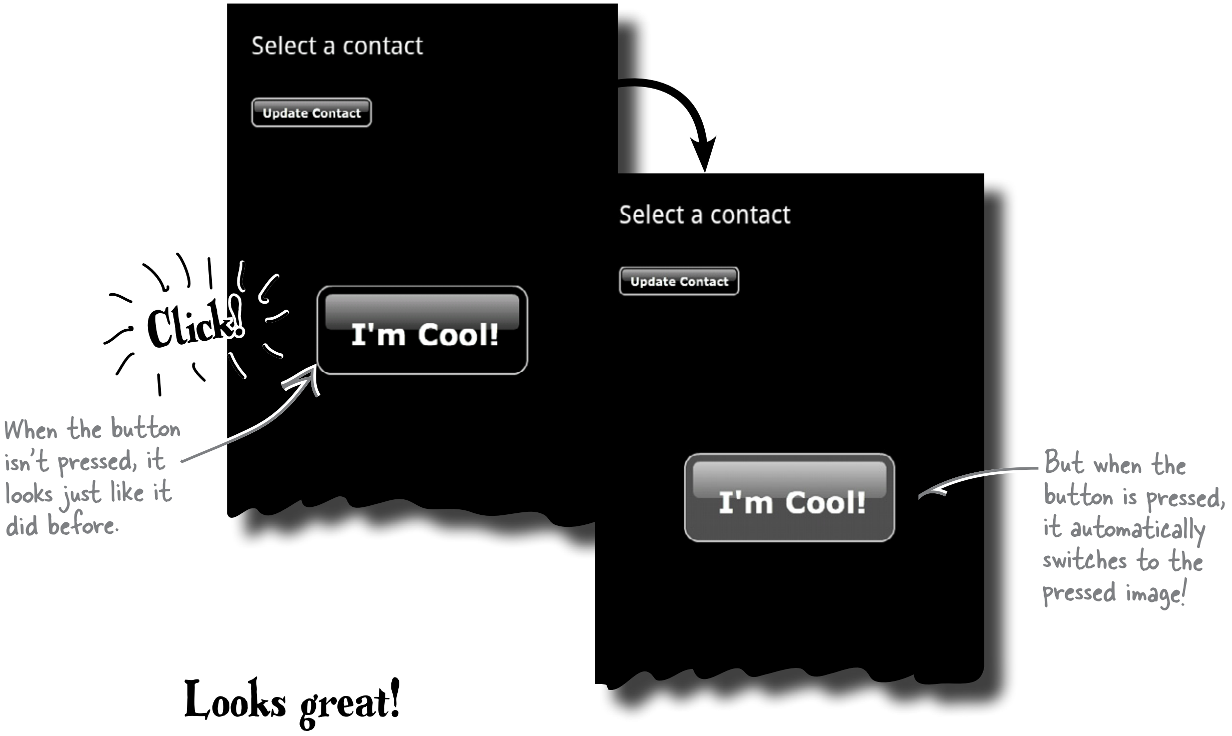
A: Pressed isn't the only state you can use for your selectors. In addition to pressed, you can also create items referencing focused, selected, checkable, checked, enabled, and window focused states. Whew, that's a lot of states! .

Q: Oh cool. But what if I want to combine then? Say I want to use one image when a button is disabled and pressed.

A: No problem! You can combine as many states as you want to in a selector item. Just add additional attributes to the item you want to configure with multiple states and you'll be all set.

Test Drive

Now that you have the selector in place, run the app and see how the "I'm Cool!" button looks when pressed and not pressed.



Add the selector for the update contact button

Now that the Button images are working for the ***I'm Cool*** button, let's add another selector for the ***Update Contact*** button. Start by adding a new selector XML file called `button_update_contact.xml`.

Do this!

Run the new Android XML file wizard again and create a new selector selector XML file called `button_update_contact.xml`.



Below is the empty `selector` for the update contact button. Add two items to that selector for the pressed and unpressed states as well. The unpressed state should point to `update_contact_btn_bkg.png` and the pressed state should point to `update_contact_btn_bkg_pressed.png`. When you're done, update the snippet from `main.xml` below to use your new selector.

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">

    ..........................................................................

    ..........................................................................

    ..........................................................................

    ..........................................................................
</selector>
```

Add items here for the unpressed and pressed button states.



button_update_contact.xml

```
<ImageButton android:id="@+id/update_contact"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Update Contact"
      android:layout_below="@+id/contact_phone"
      android:layout_alignLeft="@+id/contact_name"
      android:layout_marginTop="10dp"
      ..........................................................
      android:background="@null"
      />
```

Set the drawable to selector.



main.xml



Below is the empty `selector` for the update contact button. Add two items to that selector for the pressed and unpressed states as well. The unpressed state should point to `update_contact_btn_bkg.png` and the pressed state should point to `update_contact_btn_bkg_pressed.png`. When you're done, update the snippet from `main.xml` below to use your new selector.

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:state_pressed="false"
        android:drawable="@drawable/update_contact_btn_bkg" />
    <item android:state_pressed="true"
        android:drawable="@drawable/update_contact_btn_bkg_pressed" />
</selector>
```

Two items, for pressed and unpressed states. Just like the I'm Cool button.



button_update_contact.xml

```
<ImageButton android:id="@+id/update_contact"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Update Contact"
    android:layout_below="@+id/contact_phone"
    android:layout_alignLeft="@+id/contact_name"
    android:layout_marginTop="10dp"
    android:src="@drawable/button_update_contact"
    android:background="@null"
    />
```

Set the android:src attribute to the selector as its drawable.

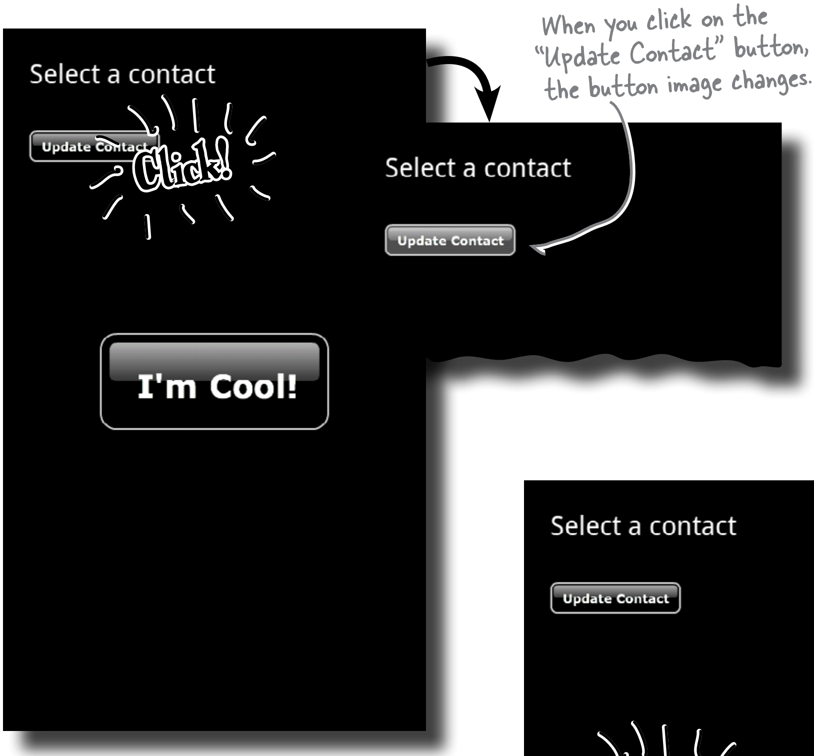


main.xml



Test Drive

Run the app now and pay close attention to the two buttons. Press and unpress the buttons a few times and watch their states go back and forth from pressed and unpressed, changing images between the two PNG files as the states change. And all you had to do was make a selector!





Great work!

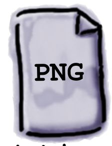
Now for the background image

The buttons are looking great, so it's time to move on to the background image. The Head First Graphics Team mocked up the background and sent along the image they used.

Do this!



Add the new file skateboard_background.png to your project.



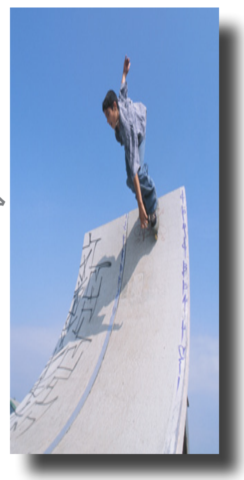
skateboard_background.png

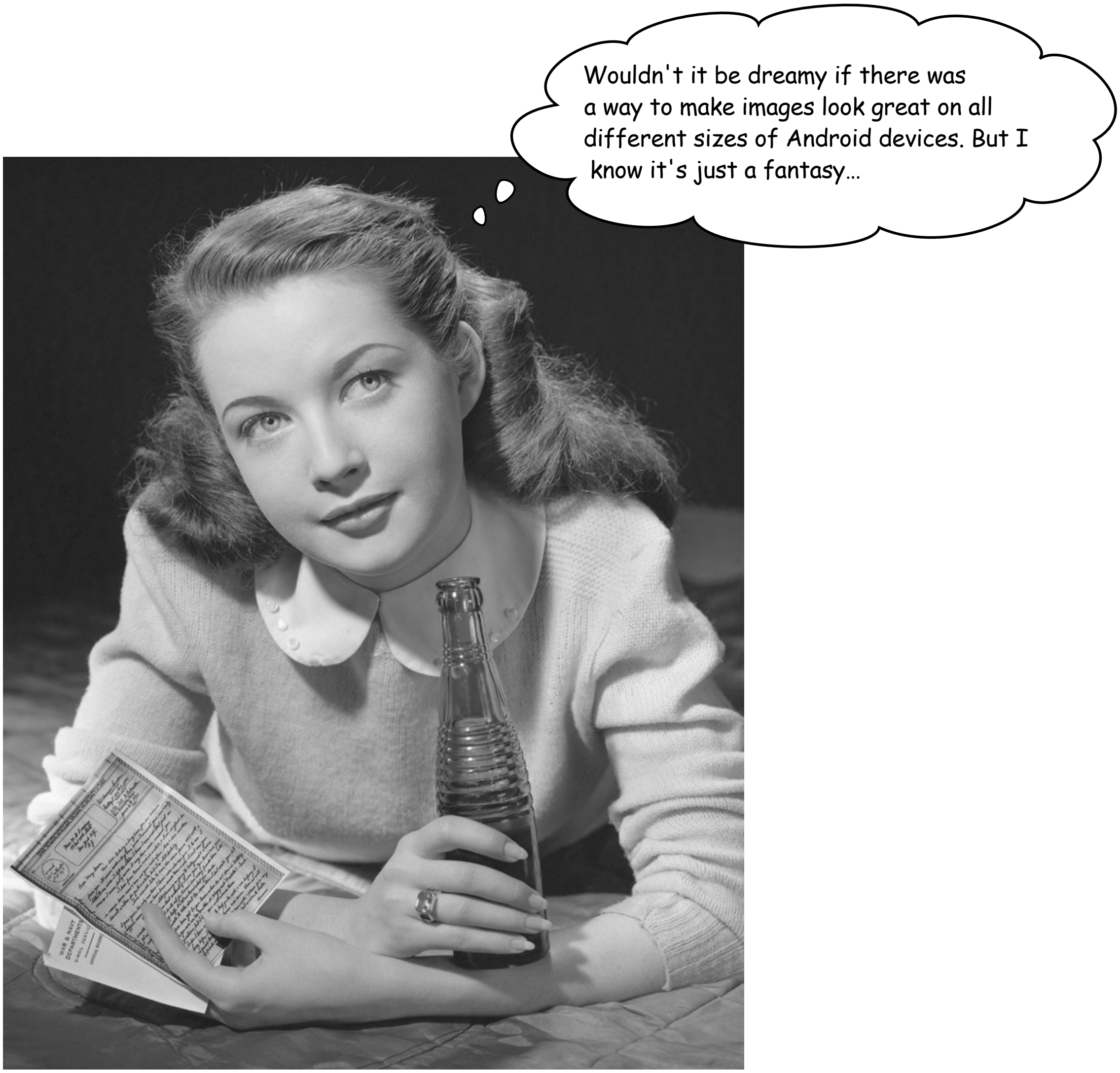
But there's a problem lurking...

The Head First Graphics Team sent you a background image that is **300x300 pixels**. But Android devices can be all kinds of different sizes! Android can resize the image, but this resizing can make your images look pretty bad with default stretching.. Just take a look:



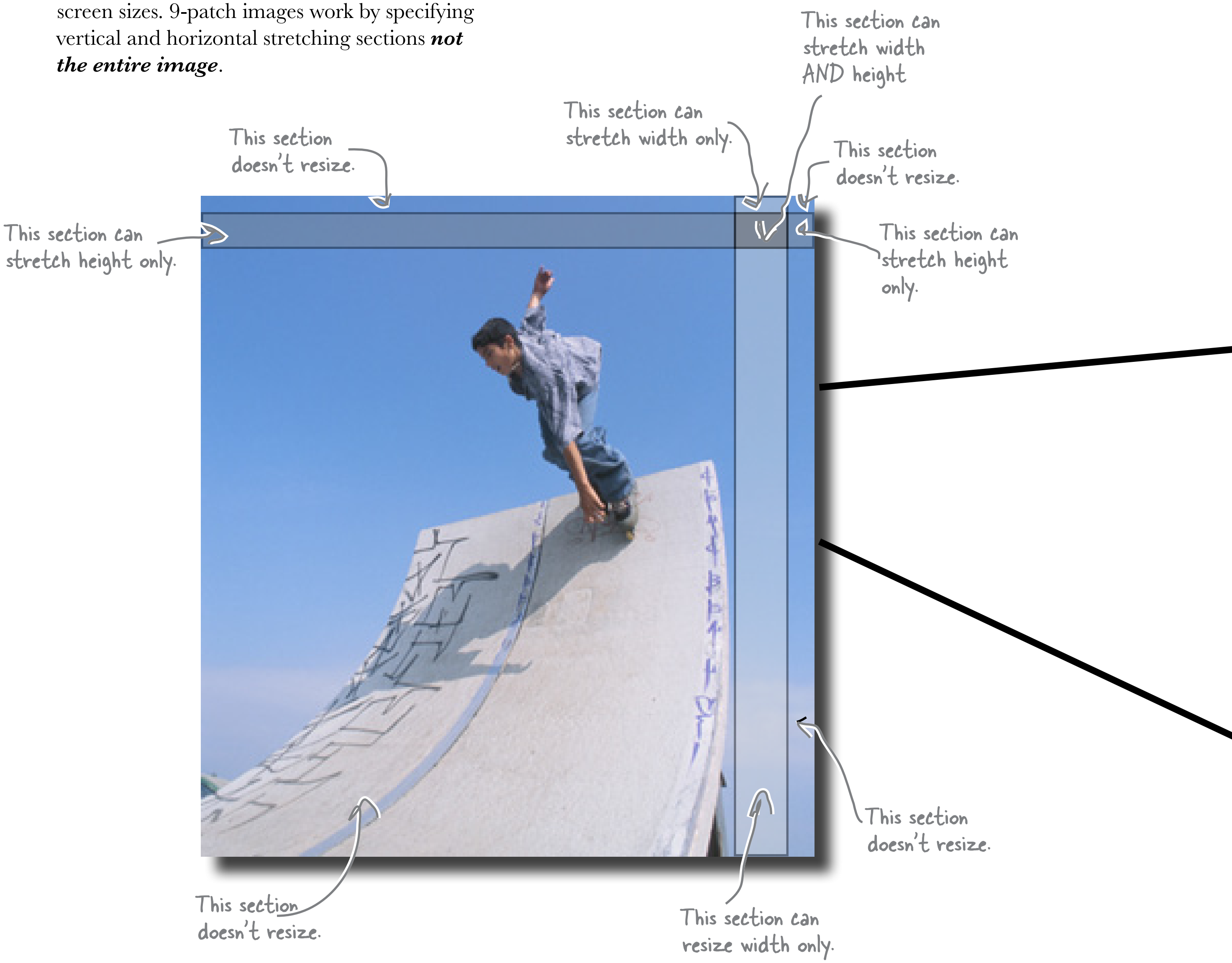
Default image resizing and can push and pull your images in ways that make them look terrible!




Wouldn't it be dreamy if there was a way to make images look great on all different sizes of Android devices. But I know it's just a fantasy...

Use 9-patch images...

You can use a technique called **9-patch images** to really help deal with these variances between screen sizes. 9-patch images work by specifying vertical and horizontal stretching sections ***not the entire image***.



Then, when the image needs to be resized, it only resizes the portions you've specified can be stretched either vertically, horizontally, or both.

... which can look great when resized!

The image can be resized as needed, but since the areas specified scale well and can be stretched, the image looks great in all of these sizes. Here are extreme stretched versions of this image as the size of the background in portrait and landscape mode.



Look how the 'sky' section was vertically stretched a lot, but since it's a part of the image that can stretch it still looks great! The cloud also horizontally stretched a little and still look great. All and all, the image is a LOT taller, but still looks sharp!

Here is the image sized for a landscape background. The vertical sky part stretched a little bit, but the clouds stretched a TON horizontally... but still looks great!

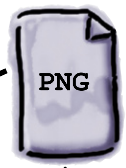


Making your own 9-patch images

Making your own 9-patch images is a snap, but you'll need to follow a little process to do it. Here is what you'll need to do.

1 **Get a raw PNG image**

9 patch images start with plain old PNG images. The only thing special about these images is that they have to resize well based



my_pic.png

2 **Edit the PNG in Draw 9-patch**

Draw 9-Patch is an application that comes with the Android SDK. Using this application, you can define the resize points.



3 **Use the 9-patch image**

Once you save a 9 patch image from Draw 9-Patch, it works just like a regular drawable that you can use in your XML files.



my_pic.9.png

The '9' before the .png lets you know it's a 9-patch image.



Choosing Images Up Close

Using 9-patch images works really well, but only for images that have a stretchable area. For this to work, you'll need a section that can be stretched horizontally, a section that can be stretched vertically, and they have to intersect.



This image has a stretchable horizontal and vertical section AND they intersect.

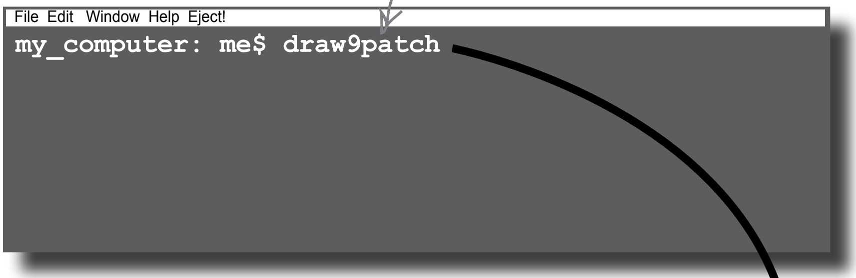
This image doesn't have any stretchable sections. Anywhere you try and stretch this image will look distorted.



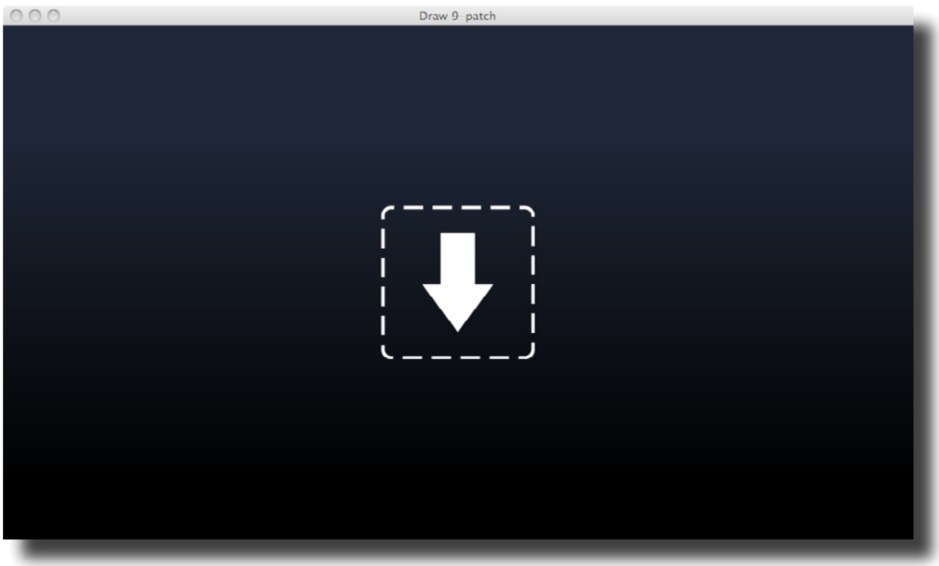
Open Draw 9-patch

The draw9patch application is location in your <android_home>/tools directory. You can launch it by typing at the command line <android_home>/tools/draw9patch.

You won't have to enter the full path to the android home if you add it to your path.

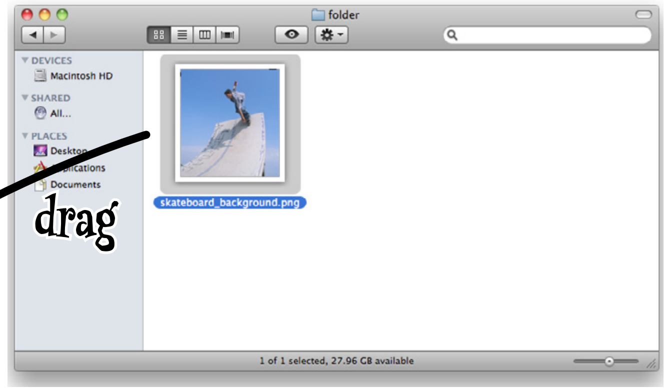


When draw9patch opens, you'll see this empty screen since there is no 9-patch image opened yet. From here, you can open a plain PNG file to create a new 9-patch image, or an existing 9-patch to edit.

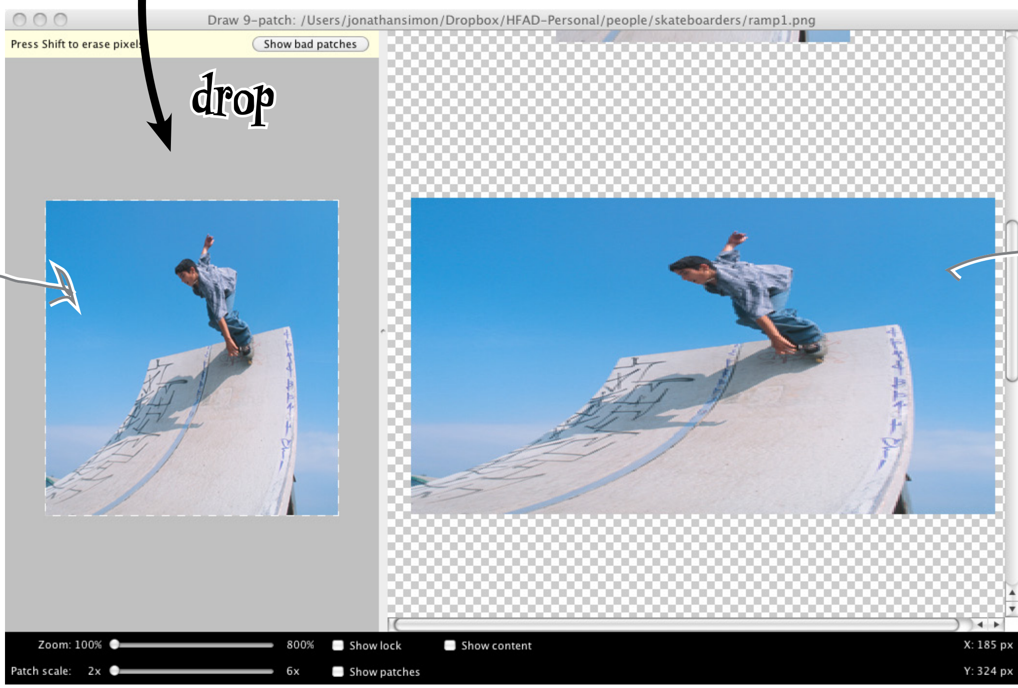


Open your PNG

Open the background image by dragging the PNG onto `draw9patch`. Since you're working on the background image, take the background image that the Head First Design Team sent you and drag it onto `draw9patch`.

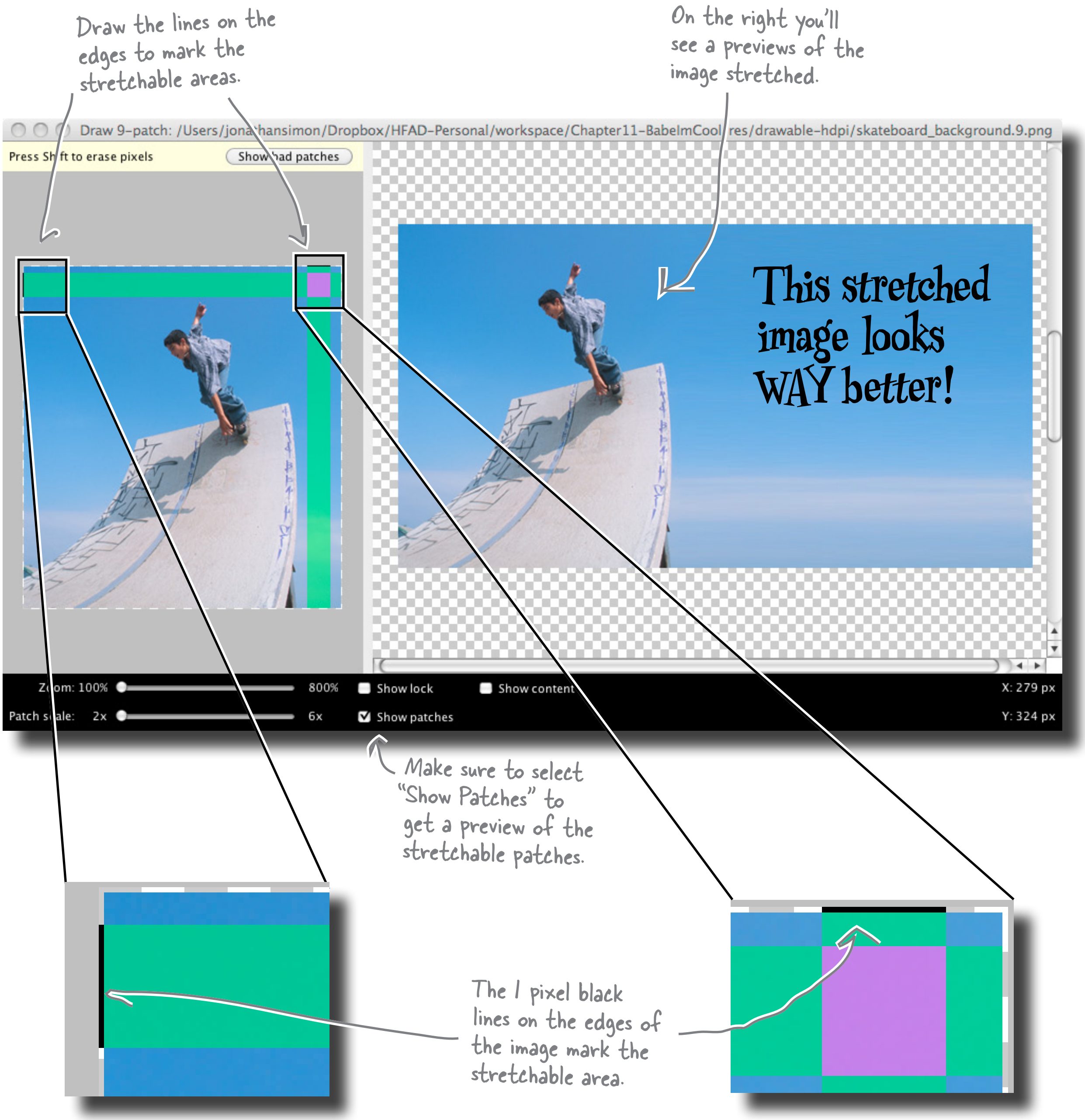


Once the image is opened in draw9patch, you'll see the image preview along with previews of the image at various different sizes.



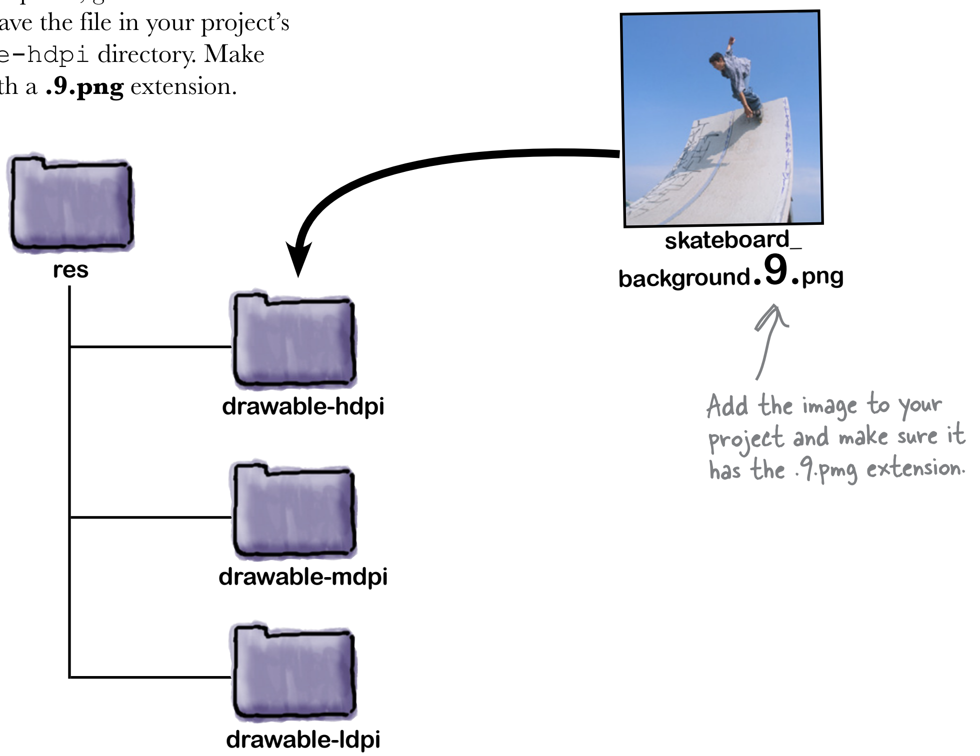
Adjust the path bounds

The path bound are what control the different patches of the 9-patch image. Draw pixels on the left, top, right and bottom edges to add to the resizing sections.



Add the 9-patch image to your project

From inside draw9patch, go to **File → Save 9-patch...** and save the file in your project's `res/drawable-hdpi` directory. Make sure to save it with a **.9.png** extension.



After you add the 9-patch image file to your project, you'll see an updated R file including a `@drawable` constant for your new 9-patch image.

Watch it!

Make sure skateboard_background.png isn't in your project when you try and save the 9-patch.

The 9-patch drawables are not unique, they are just drawables with special extensions. As far as the Android runtime is concerned, skateboard_background.png and skateboard_background.9.png are the same drawable resource (they just act different in the running app). So if you already added skateboard_background.png to your project, make sure you delete it before adding saving the 9-patch image or you'll get a nasty error!

Use the 9-patch image in your layout

Once you have the 9-patch image added to your project, you can use it like any other drawable. You can set it as the `android:src` of an `ImageView` or `ImageButton`, or the `android:background` for a other Views.



Below is the beginning of the main `RelativeLayout` for the main screen. Set the background of the layout to your new 9-patch image using the `android:background` attribute. This will set the 9-patch image as the background for the entire screen.

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        >

        <TextView android:id="@+id/contact_name"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:layout_alignParentTop="true"
              android:layout_alignParentLeft="true"
              android:layout_marginLeft="20dp"
              android:layout_marginTop="20dp"
              android:textSize="20dp"
              android:textColor="#ffffff" />
```



main.xml

Exercise Solution

Below is the beginning of the main `RelativeLayout` for the main screen. You should have set the background of the layout to your new 9-patch image using the `android:background` attribute. This will set the background for the entire screen.

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
       xmlns:android="http://schemas.android.com/apk/res/android"
       android:layout_width="fill_parent"
       android:layout_height="fill_parent"
    →  android:background="@drawable/skateboard_background" >

       <TextView android:id="@+id/contact_name"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:layout_alignParentTop="true"
              android:layout_alignParentLeft="true"
              android:layout_marginLeft="20dp"
              android:layout_marginTop="20dp"
              android:textSize="20dp"
              android:textColor="#ffffff" />
```

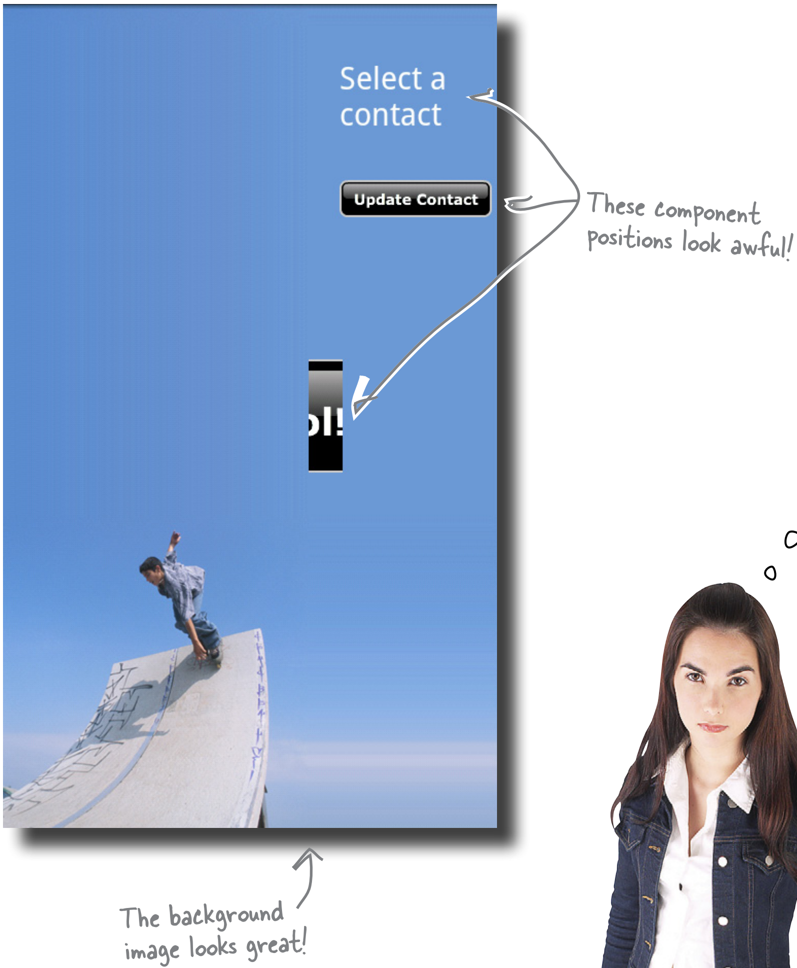
Set the android:background property to the 9-patch drawable.

XML

main.xml

Test Drive

Now that the 9-patch image is set as the background. run the app and see how it looks!





Adjust the padding

Usually padding isn't an issue with 9-patch images and you can easily use them as backgrounds for `ImageButtons` and other Views. But when you set the background of a RelativeLayout to a 9-patch image, you need to watch out for padding issues. *It's an easy fix though.* Just set the padding to 0dp and you'll be all set. This overrides any default padding the Android runtime is trying to use which was causing all of that ***crazy*** positioning.

```
<RelativeLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:background="@drawable/skateboard_background"
        android:padding="0dp"
        >
```

← Set the padding to 0dp.



main.xml

there are no Dumb Questions

Q: Can I use 9-patch images with selectors?

A: You sure can, and it's a pretty common thing to do. You can use a 9-patch image for a button background, with one for pressed and one for not pressed. Then use Android text rendering instead of using the text embedded in the image and you can use the same pressed and non-pressed images over and over again!

Q: DO I have to make separate 9-patch images for different screen densities?

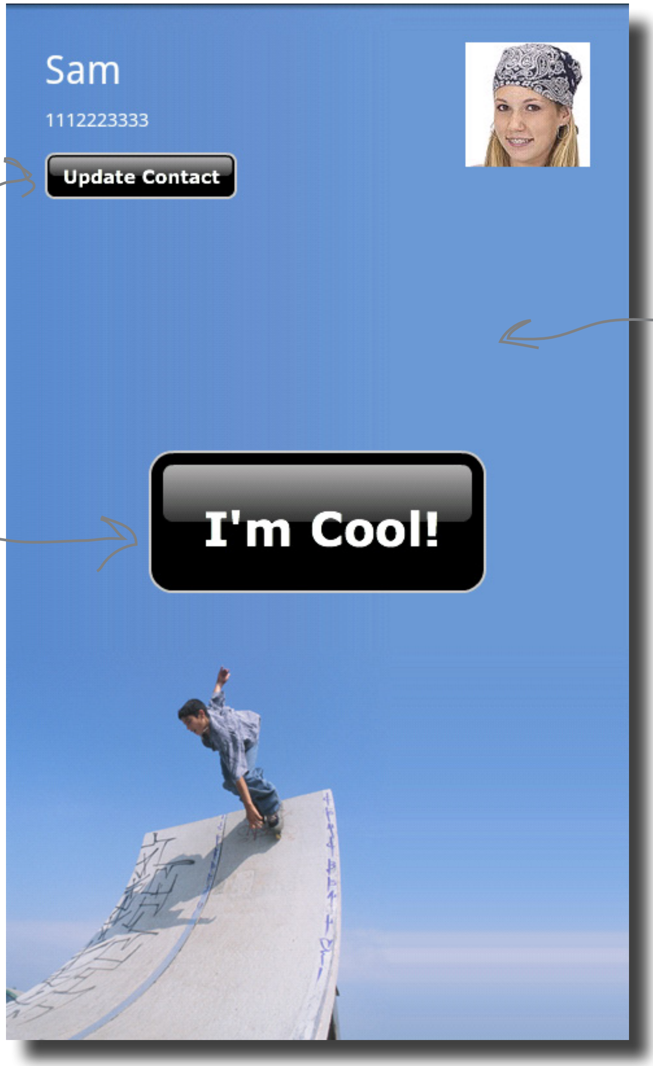
A: Yes. Like all other image resources, 9-patch images are density dependent. Since 9-patch images scale though, you can sometimes get away without it. But it's always a good idea to include multiple densities.

Q: Do the 9-patch images have to resize the same for each pixel density?

A: No. The 9-patch image includes both the image as well as the resizing areas. (The resize is marked with black pixels on the image border). So you can resize the images differently for each screen density. That said, you probably want to keep them pretty similar to keep your app consistent.

Test Drive

Try running the app again, this time with the overridden padding set to `0dp`.



The buttons are correctly positioned and look great with the images.

The background image looks great.

Way better. Looks great now!

Out in the wild

Testing the app is one thing, but the real reason you're building the app is for Sam and Scott to use it! Let's give them the app for the day and see how they use it.



That was a crazy drop! Let me send Sam a message and her late her know, I'm cool.




I'm Cool.
Snap! My Phone just buzzed. Hey, it's Sam using our new app! I bet he just did a big drop, and let me know he's OK. Man, that's so sweeeeeet!



Go Off Piste

That was some great work you did with the button graphics and 9-patch backgrounds. Here are some other things to look into if you want to make the app even better!

Use more 9-patch images

There are number of places you could use 9-patches to make the app cooler. You could use generic 9-patch images for both buttons. You could also make a cool 9-patch border for the the contact photo to make it stand out a little more.

Add location to the txt

It's cool to let someone know you're OK, but even cooler to let them know where you are too! We won't go into it here, but look into the Android location APIs and add location info to the text message the app is sending.

Save the selected contact

You probably noticed that every time you ran the app, you had to select the contact again! That's because it's not being saved to the database. Use what you're learned about Android SQLite databases to save the contact and automatically reload it on startup.



Your Android Toolbox

You just did some major graphics heavy lifting! Let's review what you've learned here that you can apply to all of your apps.

9-patch images

- Find an image that can stretch horizontally and vertically, and that those sections overlap
- Use draw9patch to mark the expandable sections
- Use the image just like any other drawable

Selectors and ImageButtons

- Add images for the states (i.e. pressed, not pressed, selected, not selected, etc)
- Create a selector XML file using the wizard
- Add items for each state and reference the image drawable to use for that state
- The selector is a 'drawable' so set the drawable source on your ImageButton to the selector

BULLET POINTS

- Use `ImageButtons` when you want to use images for your buttons.
- Set the background drawable to `@null` to remove borders.
- Use Selectors to add multiple images to a single button based on state.
- Selector XML files go in the `res/drawable` directory. You don't need a separate selector for each screen size.
- Use **9-patch** images to create expandable images
- Once you have a good PNG, use **draw9patch** to mark the resiable sections.
- Add 9-patch images to your project just like any other image drawable, in the res directory specific to your screen size.
- Make sure you have separate 9-patch images for each screen size group you are supporting.
- You can use 9-patch images for all kinds of resizable needs: background of EditTexts and TextViews, layout backgrounds, and more!

Leaving town...



It's been great having you here in Androidville!

We're sad to see you leave, but there's nothing like taking what you've learnt and putting it to use. You're just beginning your Android journey and we've put you in the driving seat. We're dying to hear how things go, so ***drop us a line*** at the Head First Labs web site, **www.headfirstlabs.com**, and let us know how Android is paying off for **YOU**!